2003

A 2-digit multidimensional logarithmic number system filterbank processor for a digital hearing aid.

Hongbo (Jennifer). Li
University of Windsor

A 2-Digit Multidimensional Logarithmic Number System Filterbank Processor for a Digital Hearing Aid

by

Hongbo (Jennifer) Li

A Thesis
Submitted to the Faculty of Graduate Studies and Research through the Department of Electrical and Computer Engineering in partial fulfillment of the requirements for the Degree of Master of Applied Science at the University of Windsor

Windsor, Ontario, Canada

2003

Your file Votre référence
ISBN: 0-612-82885-9
Our file Notre référence
ISBN: 0-612-82885-9

Abstract

This thesis addresses the design, implementation, and evaluation of a filterbank for digital hearing aids using a Multidimensional Logarithmic Number System (MDLNS). Hearing loss is a function of both frequency and input level. In a typical digital hearing instrument, the hearing loss compensation is performed by separating the incoming sound into several frequency bands which are then compressed to allow the amplification of low level signals while maintaining the amplitude of high level signals.

The demands of low power consumption and small size have led to a number of advances in algorithms, semiconductor technologies and system architectures for completely-in-canal (CIC) hearing aid device. Based on research for digital hearing aids that started in the early 1990's, we have developed a new number system (MDLNS) and associated architecture that benefit the digital hearing aid processor in both of these requirements. Although the LNS has been previously considered for digital hearing aid processors, this thesis presents an exploration of the MDLNS for digital hearing aid circuitry. As with the LNS, the MDLNS provides a reduction in the size of the number representation, but the MDLNS promises a lower cost (*area.power*) implementation of the arithmetic operations required in both the linear and non-linear domains of filtering and compression. In this thesis we discuss an application of the MDLNS on the construction of a finite impulse response FIR filterbank, a major component of digital hearing aid processors. The MDLNS filterbank processor chip was fabricated using a 0.18 micron CMOS technology.

After evaluating the MDLNS filterbank and the two state-of-the-art filterbanks using classic binary implementation, we found power, area, and performance of the MDLNS filterbank processor showed competitive results compared to those binary filterbank processors.

To my parents who taught me not to fear failure, the determination to succeed and the courage to try. To my husband Norman and my son Barton for their support.

Acknowledgments

I would like to express my deepest thanks and appreciation to my supervisor Dr. G. A. Jullien for his continuous guidance, support and encouragement and for bringing this challenging project to my attention. I am grateful to Gennum Corp. for providing funding.

I would also like to thank my committee members Dr. Majid Ahmadi, Dr. Vassil Dimitrov, Dr. R. G. Gaspar and Dr. Chunhong Chen for their interest and invaluable comments.

Finally, I would like to express my thanks to Roberto Muscedere for his aid with HDLs, Cadence tools and helping in producing the MDLNS filterbank microchip.

The CAD tools used in the project were provided though CMC.

Table of Contents

Abstract....iv

Acknowledgments....vi

List of Symbols....x

List of Abbreviations....xi

List of Figures....xii

List of Tables....xv

Chapter 1 Introduction....1

1.1 Introduction....1

1.2 Human Hearing....2

1.2.1 Dynamic Range....2

1.2.2 Frequency Range....3

1.3 Hearing Loss....5

1.3.1 Conductive Hearing Loss....6

1.3.2 Sensorineural Hearing Loss....6

1.4 Hearing Instrument Types....10

1.5 Hearing Instrument Technologies....12

1.6 MDLNS....16

1.7 Thesis Organization....19

Chapter 2 MDLNS Background Theory....20

2.1 Introduction....20

2.2 Representation....21

2.3 Input Data Mapping....24

2.3.1 Error-Free Representations....26

2.3.2 Non Error-Free Representations....28

2.3.3 Mathematical Operations....29

2.4 Hardware Complexity....32

2.4.1 Single Digit Computational Unit....32

2.4.2 n-digit Computational Unit....33

2.4.3 MDLNS Complexity Comparison....34

Chapter 3 Digital Filterbank Algorithms....36

3.1 Introduction 36
3.2 Design Constraints 36
3.3 IFIR Filterbank Algorithm Design 38
3.3.1 Introduction 38
3.3.2 Interpolated FIR filters 39
3.3.3 Complementary halfband linear-phase FIR filters 39
3.3.4 Prototype filter $H1(z)$ 40
3.3.5 Sub-filterbanks $Hs1$ and $Hs2$ 42
3.3.6 IFIR filterbank with 60 dB noise level 44
3.4 DFT Filterbank Algorithm 46
3.4.1 Oversampled WOLA Filterbank [22] 48
3.4.2 Prototype Nyquist Eigenfilter 50
3.5 MDLNS 8 Parallel FIR Filters 52
3.5.1 MDLNS Filterbank Algorithm Design 52
3.5.2 Coefficient MDLNS Mapping 55
3.5.3 Conclusions 58

Chapter 4 Filterbank Processor Architectures 59
4.1 IFIR Filterbank Architecture 59
4.2 DFT Filterbank Architecture [24] 61
4.3 MDLNS Filterbank Architecture 63
4.3.1 Improved MDLNS Architecture 65
4.3.2 Results and Conclusions 66

Chapter 5 Physical Implementation of Filterbank Processor 67
5.1 Introduction 67
5.2 Design Flow 68
5.2.1 RTL Simulation 69
5.2.2 Logic Synthesis 72
5.2.3 Gate-Level Simulation 76
5.2.4 Floorplanning and Timing-Driven Placement 76
5.2.5 Clock Tree Generation 77
5.2.6 Pre-Route Golden Verilog Netlist Verification 78
5.2.7 Routing 79
5.2.8 Post-Layout Static Timing Analysis 79
5.2.9 Physical Verification (DRC & LVS) 79
5.3 MDLNS Filterbank Processor Chip 80

Chapter 6 MDLNS Filterbank Processor Analysis and Comparison 82
6.1 Introduction 82
6.2 Quantization Noise Comparison 82
6.3 Area Comparison 95
6.4 Power Comparison 96

Chapter 7 Conclusion and Future Works 98
7.1 Conclusion 98
7.2 Contributions 99
7.3 Suggestions for Future Work 100

REFERENCES 101

Appendix A Matlab Code for Filterbanks 104
A.1 Introduction 104

A.2 Matlab Code for DFT Filterbank 104
A.3 Eight 60 dB Parallel FIR Filters 107

Appendix B Verilog Code for MDLNS Filterbank 120
B.1 Testbench.v 120
B.2 parameters.v 122
B.3 chipfilterbank.v 123
B.4 Binary to MDLNS converter 145

Appendix C Matlab Code for MDLNS and Binary Filterbank Comparison 161

Appendix D MDLNS Filterbank Chip Testing 167
B.1 Test Overview 167
B.2 Test Procedure 169

List of Symbols

m	*mantissa*
r	*base or radix*
Φ	*Look-up table operator for MDLNS addition*
Ψ	*Look-up table operator for MDLNS subtraction*
t	*MDLNS ternary exponent*
b	*MDLNS binary exponent*

List of Abbreviations

ASIC	*Application Specific Integrated Circuits*
CAD	*Computer Aided Design*
CMC	*Canadian Microelectronics Corporation*
CMOS	*Complementary Metal-Oxide Semiconductor*
CT-GEN	*Clock Tree Generator*
dB	*Decibels*
DEF	*Design Exchange Format*
DRC	*Design Rule Check*
FIR	*Finite Impulse Response*
HDL	*Hardware Description Language*
IC	*Integrated Circuit*
LEF	*Library Exchange Format*
LNS	*Logarithmic Number System*
LVS	*Layout Versus Schematic*
MAC	*Multiply Accumulate*
MDLNS	*Multi-dimensional Logarithmic Number System*
RAM	*Random Access Memory*
RC	*Resistance and Capacities*
ROM	*Read Only Memory*
RTL	*Register Transfer Level*

List of Figures

Figure 1.1 Frequency and intensity ranges of speech and music 5
Figure 1.2 Age-induced hearing loss....7
Figure 1.3 Noise induced hearing loss 8
Figure 1.4 Ski-slope hearing loss 9
Figure 1.5 (left) Meniere's Disease (low_frequency loss), right: Trauma (mid_frequency loss)....9
Figure 1.6 BTE type of hearing aid....10
Figure 1.7 ITE type of hearing aid 10
Figure 1.8 ITC type of hearing aid....11
Figure 1.9 CIC Hearing instrument placement 12
Figure 1.10 Digital hearing instrument structure [1]....14
Figure 1.11 An example of the DSP part of the hearing aid block diagram 14
Figure 1.12 Compressive amplification....16
Figure 1.13 Signal-to-quantization noise ratio vs. input level....18
Figure 1.14 Filter coefficient distribution....19
Figure 2.1 A 2-digit DBNS representation of 79 25
Figure 2.2 A 2DLNS representation of 7.25 26
Figure 2.3 Absolute error (>0.5) for different second bases 28
Figure 2.4 1-digit MDLNS inner product computation unit 32
Figure 3.1 a) Lowpass modelfilter b) Interpolated model filter c) Complementary interpolated model filter....40
Figure 3.2 Realization of an interpolated complementary linear-phase FIR filter....41
Figure 3.3 Filterbank structure with prototype filter and subfilterbank 42
Figure 3.4 a) Sub-filterbank Hs1 b) Sub-filterbank Hs2 43
Figure 3.5 IFIR filterbank structure....44
Figure 3.6 Magnitude response of 60 dB IFIR filterbank 45
Figure 3.7 Maximally flat overall magnitude response of 60 dB IFIR filterbank....45
Figure 3.8 Complex modulator model of the DFT filterbank 47

Figure 3.9 Frequency response of DFT filterbank channels for odd and even channel stacking arrangements....49
Figure 3.10 Overall magnitude response of the DFT filter bank....49
Figure 3.11 Frequency response of Eigenfilter and Kaiser Window FIR Filter....52
Figure 3.12 Magnitude response of 8 parallel filters....54
Figure 3.13 Phase response of the 8 parallel FIR filter....54
Figure 3.14 Overall magnitude response of 8 parallel filters....55
Figure 3.15 Magnitude response of MDLNS filterbank(5bit 3bit,5bit 3bit)....56
Figure 3.16 Phase response of MDLNS filterbank....57
Figure 3.17 Flat overall magnitude response of MDLNS filterbank....57
Figure 4.1 IFIR Filterbank Architecture....60
Figure 4.2 The address sequence for filter H7 (indicating the address pairs for the dual-port RAM)....61
Figure 4.3 WOLF DFT analysis processing....62
Figure 4.4 WOLA DFT filterbank synthesis processing....63
Figure 4.5 8-bank 2-DLNS filter architecture....64
Figure 4.6 Improved MDLNS processor....65
Figure 5.1 Digital ASIC Design Method....68
Figure 5.2 Hierarchal diagram for HDL description of a MDNS filterbank....69
Figure 5.3 Verilog simulation of the entire MDLNS filterbank....72
Figure 5.4 Gate level input data serial to parallel converter block (128 cells)....74
Figure 5.5 Gate level binary to MDLNS converter block (762 cells)....74
Figure 5.6 Gate level MDLNS filterbank block (1883 cells)....75
Figure 5.7 Gate level output data parallel to serial converter block (69 cells)....76
Figure 5.8 Design of MDLNS filterbank processor after placement....77
Figure 5.9 Design of MDLNS filterbank processor after CTGen....78
Figure 5.10 Design of MDLNS filterbank processor after routing....79
Figure 5.11 Block graph of MDLNS filterbank chip....80
Figure 5.12 Layout of the MDLNS filterbank chip....81
Figure 5.13 Micrograph of the fabricated MDLNS filterbank processor chip....81
Figure 6.1 Quantization noise of 16 bit binary system: channel one....83
Figure 6.2 Quantization noise of 16 bit binary system: channel 2....83
Figure 6.3 Quantization noise of 16 bit binary system: channel 3....84
Figure 6.4 Quantization noise of 16 bit binary system: channel 4....84
Figure 6.5 Quantization noise of 16 bit binary system: channel 5....85
Figure 6.6 Quantization noise of 16 bit binary system: channel 6....85
Figure 6.7 Quantization noise of 16 bit binary system: channel 7....86
Figure 6.8 Quantization noise of 16 bit binary system: channel 8....86
Figure 6.9 Quantization noise of MDLNS filterbank channel 1 (input 5 4 5 4)....87
Figure 6.10 Quantization noise of MDLNS filterbank channel 2 (input 5 4 5 4)....87
Figure 6.11 Quantization noise of MDLNS filterbank channel 3 (input 5 4 5 4)....88

Figure 6.12 Quantization noise of MDLNS filterbank channel 4 (input 5 4 5 4) 88
Figure 6.13 Quantization noise of MDLNS filterbank channel 5 (input 5 4 5 4) 89
Figure 6.14 Quantization noise of MDLNS filterbank channel 6 (input 5 4 5 4) 89
Figure 6.15 Quantization noise of MDLNS filterbank channel 7 (input 5 4 5 4) 90
Figure 6.16 Quantization noise of MDLNS filterbank channel 8 (input 5 4 5 4) 90
Figure 6.17 Quantization noise of MDLNS filterbank channel 1 (input 5 5 5 5) 91
Figure 6.18 Quantization noise of MDLNS filterbank channel 2 (input 5 5 5 5) 91
Figure 6.19 Quantization noise of MDLNS filterbank channel 3 (input 5 5 5 5) 92
Figure 6.20 Quantization noise of MDLNS filterbank channel 4 (input 5 5 5 5) 92
Figure 6.21 Quantization noise of MDLNS filterbank channel 5 (input 5 5 5 5) 93
Figure 6.22 Quantization noise of MDLNS filterbank channel 6 (input 5 5 5 5) 93
Figure 6.23 Quantization noise of MDLNS filterbank channel 7 (input 5 5 5 5) 94
Figure 6.24 Quantization noise of MDLNS filterbank channel 8 (input 5 5 5 5) 94
Figure 6.25 Micrograph of the IFIR chip 96
Figure D.1 Overview of the test setup 168
Figure D.2 Testing lab for MDLNS Filterbank Processor Chip 168
Figure D.3 List of hardware connections 171
Figure D.4 Timing cycle of MDLNS filterbank chip testing 172
Figure D.5 Device Under Test Board 173
Figure D.6 Testing chip and DUT board 174
Figure D.7 Current measurement interconnections 176
Figure D.8 Current of power supply display 176

List of Tables

Table 1.1	Sound examples for dynamic range of human hearing	3
Table 1.2	Octave bands of human hearing and their characteristics	4
Table 1.3	Hearing loss degrees and their effects	6
Table 2.1	Classical LNS and 2D LNS Comparison	35
Table 3.1	IFIR filterbank coefficients not equal to zero	42
Table 6.1	Synospys Power Compiler results of MDLNS filterbank processor	97
Table D.1	pin-to-pod mapping	174

Chapter 1
Introduction

1.1 Introduction

In this thesis, research on current digital filterbank algorithms, architectures, and hardware implementation for CIC (completely in the canal) digital hearing instruments is carried out. The design and implementation of a filterbank for digital hearing aids using the Multidimensional Logarithmic Number System (MDLNS) is carried out.

The history of hearing instruments can be traced from the 19th century on, but it is known that Greeks used shells and Romans had bronze funnels as hearing instruments. In 1800 the first hearing instrument company, which was manufacturing tubes and trumpets as hearing instruments, was established. In the mid 1890's the first patent for an electrical hearing aid was filed but the invention never reached production. 1899 was the year that the first commercially manufactured hearing aid, Akoullalion, made of carbon, was produced. The first vacuum tube hearing aid, which was very heavy and worn on the body, was patented in 1921. It consisted of a microphone, an earphone, an amplifier and two batteries, which would last only one day. With the commercial introduction of transistor technology, after 1953, the size of the hearing aids became much smaller and their capabilities were increased. In 1970 the first hybrid hearing aids, which had analog and digital circuitry were manufactured. In 1988 many programmable hearing aids were introduced; they used analog circuitry, which was programmed, digitally, with computers. In 1996 the first behind the ear 100% digital hearing instrument was manufactured [1].

1.2 Human Hearing

1.2.1 Dynamic Range

The dynamic range of human hearing is defined as the difference between the audible weakest sound and the highest comfortable level, where discomfort starts. The nominal threshold of hearing (P_0) at 1kHz is measured as $2 \cdot 10^{-5}$ Pa. In order to derive the intensity of this pressure, we first define acoustic resistance. Acoustic resistance, *r*, varies with temperature because of its dependence on the density of air, ρ, and the speed of sound, *s*, according to Eqn. (1.1). At 0 °C the acoustic resistance is 428 Rayls,

$$r = \rho \cdot s \tag{1.1}$$

whereas at room temperature it is reduced to 415 Rayls. To achieve a standard, the intensity threshold, I_0, at room temperature, is derived:

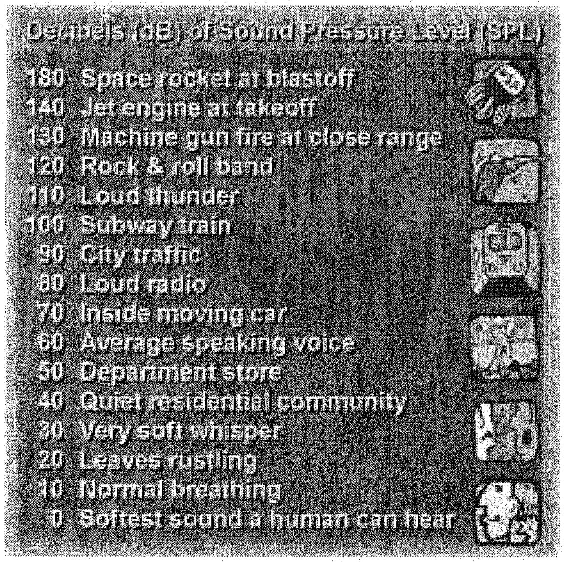
$$I_0 = (P_0)^2/r = (2 \cdot 10^{-5})^2/415 = 0.964 \cdot 10^{-12} \approx 10^{-12} Watts/m^2 \tag{1.2}$$

10^{-12} Watts/m^2 is defined as the weakest audible sound intensity level at 1kHz. The intensity or pressure of a sound signal is defined as the ratio of its value to these threshold values. The sound pressure level (SPL) and sound intensity level (IL) equations are:

$$SPL = 20 \cdot \ln(p/p_0) \qquad IL = 10 \cdot \ln(I/I_0) \tag{1.3}$$

The dynamic range of human hearing is accepted as from 0 dB SPL ($2 \cdot 10^{-5}$ Pa) to 120 dB SPL (20 Pa). Over 120 dB SPL sound causes discomfort and damage in our ear [1]. Table 1.1 shows some example sounds with their SPL and intensity values.

Actually human hearing is not only intensity dependent. The weakest audible sound intensity differs according to the frequency of that sound. For example, the weakest audible sound at 10 kHz is 20 dB SPL but the weakest audible sound at 100 Hz is 40 dB SPL [1].



Decibels (dB) of Sound Pressure Level (SPL)	
180	Space rocket at blastoff
140	Jet engine at takeoff
130	Machine gun fire at close range
120	Rock & roll band
110	Loud thunder
100	Subway train
90	City traffic
80	Loud radio
70	Inside moving car
60	Average speaking voice
50	Department store
40	Quiet residential community
30	Very soft whisper
20	Leaves rustling
10	Normal breathing
0	Softest sound a human can hear

Table 1.1 Sound examples for dynamic range of human hearing

1.2.2 Frequency Range

The frequency range of human hearing is generally accepted to be 20 Hz to 20 kHz. The cochlea splits the sound signal approximately into octave bands (a factor of two increase in frequency), thus human hearing covers approximately 10 octaves. The bands and their characteristics are given in Table 1.2 [1] . Each octave gives different characteristics to the sound signal. Because of the logarithmic frequency distribution of human hearing, the bandwidths of octave bands are increasing along the frequency axis. This shows that different, but equal amount of, audio information is located between 20 Hz - 40 Hz and 10 kHz- 20 kHz. Therefore if we omit the 10 th octave of the sound signal, the signal will still carry 90% of its information [1].

.

Octave	Frequency Range	
1st	20-40 Hz	Low Bass - These frequencies add fullness, power and boom to sound. Lowest notes of bass, piano and tuba fall into this category.
2nd	40-80 Hz	
3rd	80-160 Hz	Upper Bass - These frequencies provide a balance in the structure of sound. Without them, sound is thin. The lower tones of the cello, trombone and rhythm sections produce sounds in this range.
4th	160-320 Hz	
5th	320-640 Hz	Midrange - Sounds get their intensity from this range of frequencies. Fundamentals and lower harmonics of most sound sources fall into this category.
6th	640-1280 Hz	
7th	1280-2560 Hz	
8th	2560-5120 Hz	Upper Midrange - Humans hear this range of frequencies best. 3000-3500 Hz contains information, which improves the intelligibility of speech and lyrics. If this band is incorrectly processed, sound becomes unpleasant. Frequencies above 3500 Hz give sound realism and clarity. Listeners perceive sound in this section of the octave (and up to about 6000 Hz in the 9th octave) as being close. Thus, 3500-6000 Hz is known as the presence range.
9th	5120-10240 Hz	Treble - Frequencies in this range give sound sparkle and brilliance. Most humans do not hear much beyond 16000 Hz
10th	10240-20480 Hz	

Table 1.2 Octave bands of human hearing and their characteristics

Male speech lies between 100 Hz and 8 kHz, whereas the harmonics of female speech can reach up to 10 kHz. The frequency and dynamic ranges of speech and music are plotted in Figure 1.1 [1]. The solid line shows the border of normal hearing. We observe that human speech can be totally extracted if the original signal is low-pass filtered up to 8 kHz. On the other hand music covers a wider frequency band than does speech [1].

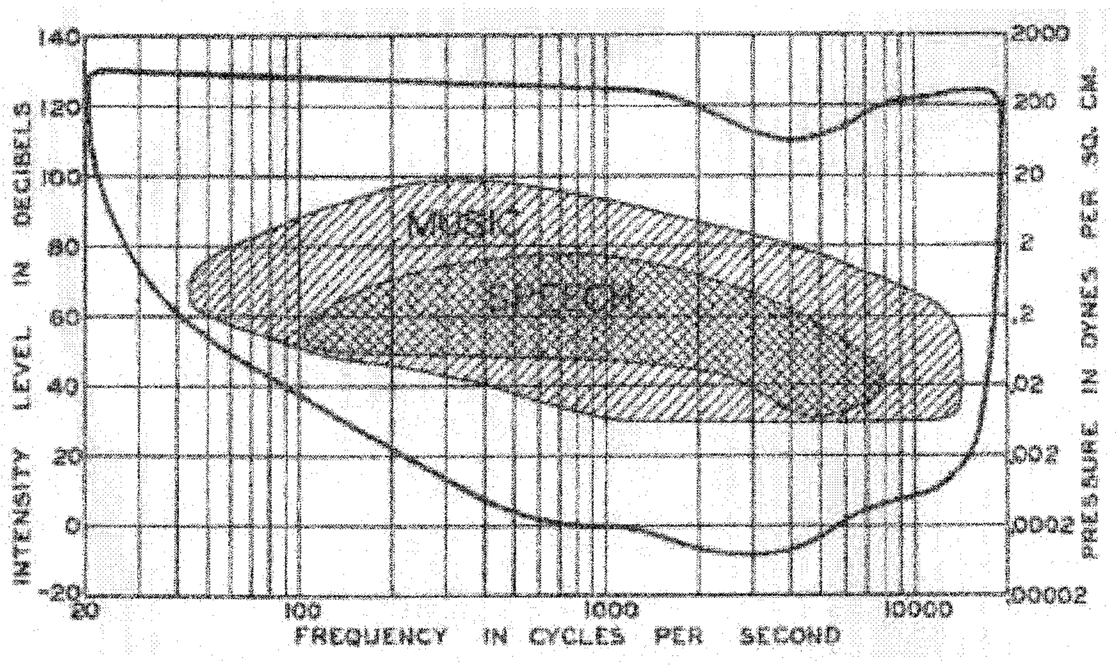
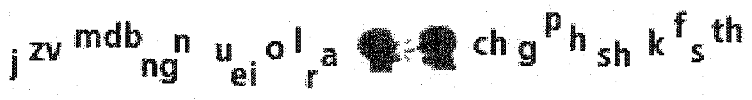


Figure 1.1 Frequency and intensity ranges of speech and music

The letters shown in the figure below symbolize sounds in our language. They are spread across frequencies and intensities. The consonants are critical for speech understanding. Note how they are centered in the high frequency area where most hearing losses occur [3].



1.3 Hearing Loss

The intricate structure of the human ear can be damaged by different sources. The degrees of hearing losses and their effects are given in Table 1.3 [1]. Hearing Loss is generally separated into two categories: 'Conductive Hearing Loss' and ‘Sensorineural Hearing Loss’.

Table 1.3 Hearing loss degrees and their effects

LOSS	CLASSIFICATION	EFFECTS
0-15 dB SPL	Norman hearing	-
16-25 dB SPL	Borderline normal (children)	-
15-25 dB SPL	Slight	Minimal difficulty with soft speech
25-40 dB SPL	Mild	Difficulty with soft speech
40-45 dB SPL	Moderate	Frequent difficulty with normal speech
46-70 dB SPL	Moderate-severe	Occasional difficulty with loud speech
71-90 dB SPL	Severe	Frequent difficulty with loud speech
>91	Profound	Near total loss of hearing

1.3.1 Conductive Hearing Loss

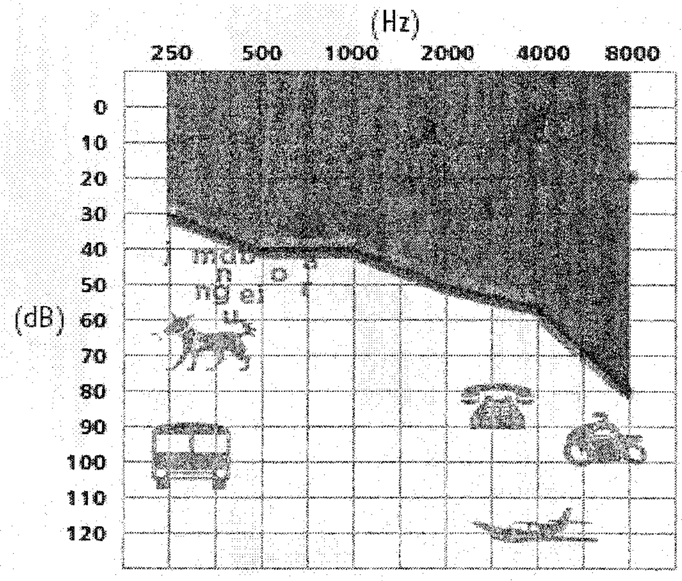
Conductive hearing loss is caused by damage to, or a malfunction of, the outer and middle ear. The reasons for this loss include: abnormality of the outer ear, ossicular discontinuity, middle ear infection, otosclerosis, perforated eardrum, earwax. It is because of the fact that the inner ear is functioning properly, that conductive hearing losses can be treated either medically or surgically [3].

1.3.2 Sensorineural Hearing Loss

This type of hearing loss covers disorders of the inner ear and the auditory neural system. Commonly the problem is due to the damage of the haircells inside the cochlea. The hearing can diminish gradually or suddenly (as with a trauma). The result is permanent hearing loss, which can not be cured with medicine or by surgery [1]. 80% of the patients who are suffering from hearing loss have sensorineural loss. Therefore, hearing instrument designers focus on this portion of hearing loss.

Causes and characteristics of sensorineural hearing loss [3]

Aged induced - or presbycusis: All human beings suffer a loss of hearing as part of the aging process. Gradual deterioration of hair cells in the inner ear can lead to a significant hearing loss. All frequencies may be affected, but it is mainly the loss of the high frequency area that results in difficulty in understanding speech, as shown in Figure 1.2. It also shows us that aging causes more hearing loss for males than for females.



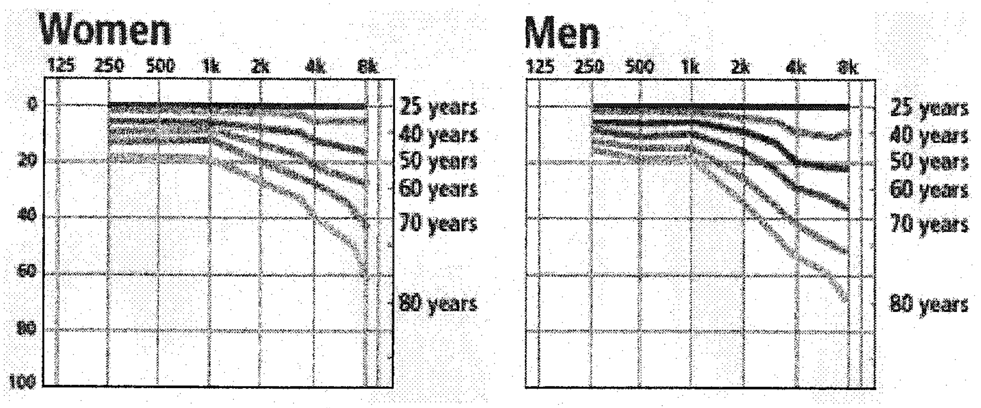


Figure 1.2 Age-induced hearing loss

Noise induced hearing loss:

An acoustic trauma or having been subjected to excessive noise for extended periods of time causes 'noise induced' hearing loss. These patients will loose some high frequency sounds as shown in Figure 1.3, but it is primarily the loss of hearing sensitivity that makes listening difficult in heavy background noise.

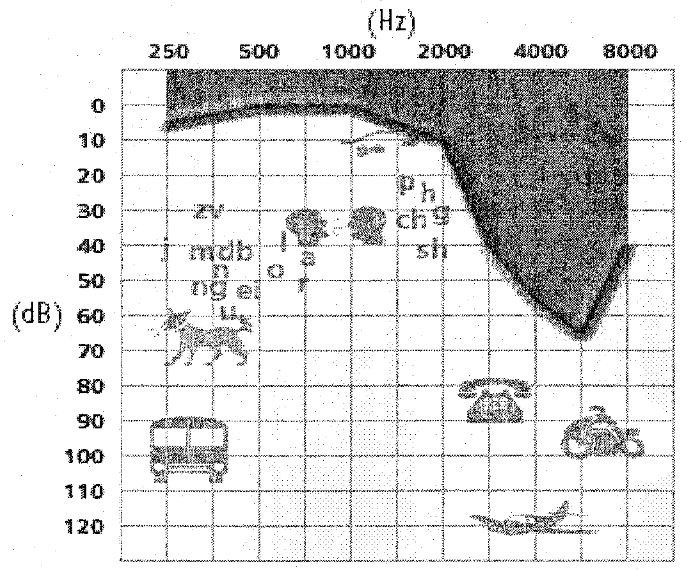


Figure 1.3 Noise induced hearing loss

Ski-slop hearing loss:

The causes of this type of hearing loss are one or more of the following: acoustic trauma, extended periods of time spent in excessive noise, lack of oxygen during birth, virus infection, genetic defects, a severe side effect of medication. A ski-slope hearing loss is often a combination of loss of inner and outer hair cells. The effect is a significant loss of high frequency sounds as shown in Figure 1.4.

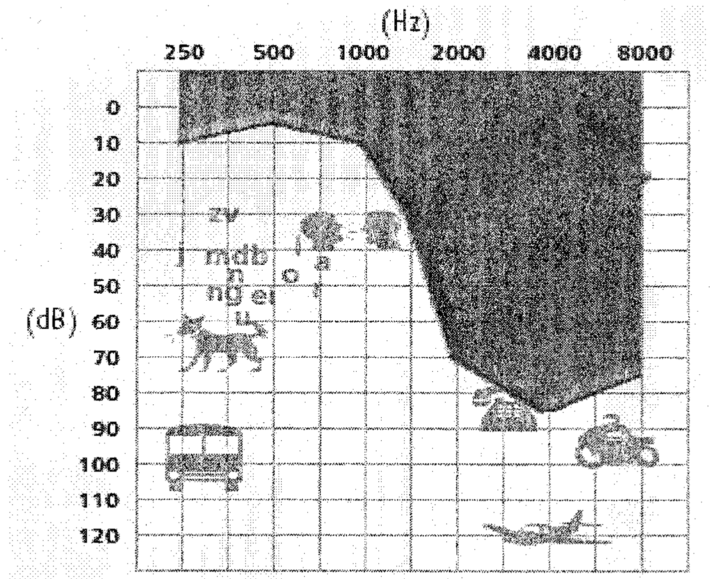


Figure 1.4 Ski-slope hearing loss

Meniere's Disease represents a typical low-frequency hearing impairment. Trauma can also causes mid-frequency loss, as shown in Figure 1.5 [2].

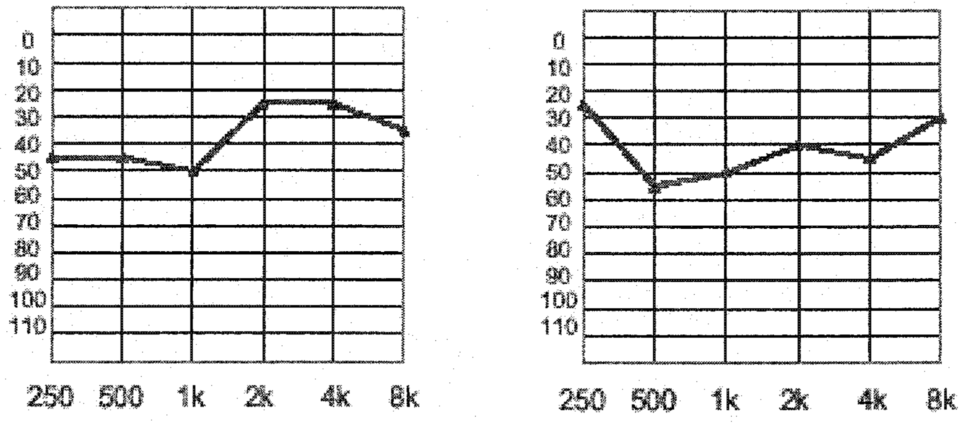


Figure 1.5 (left) Meniere's Disease (low_frequency loss), right: Trauma (mid_frequency loss)

1.4 Hearing Instrument Types

Hearing instruments can be grouped, according to their sizes, into four categories: BTE (behind the ear), ITE (in the ear), ITC (in the canal) and CIC (completely in the canal) [1].

BTE (Behind the ear): These hearing instruments rest behind the earlobe and connected to the ear by a custom earmold. Their location allows a relaxation in their size constraint. Therefore circuits allowing more gain can be used. BTE instruments can be used to correct a wide range of hearing losses from mild to profound.

Figure 1.6 BTE type of hearing aid [3]

ITE (In the ear): These instruments fit in the bowl of the ear and are visible in the ear. They are also powerful based on their size and can be used for hearing losses from mild to severe.

Figure 1.7 ITE type of hearing aid [3]

ITC (In the canal): These hearing instruments are located further into the canal than the ITE types, but are still visible. They can be used for hearing losses from mild to moderate.

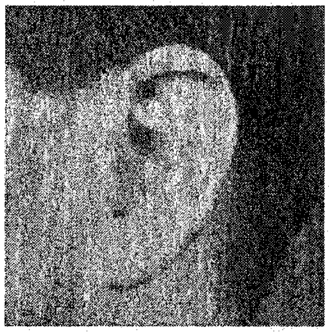


Figure 1.8 ITC type of hearing aid [3]

CIC (completely in the canal): These instruments go very deep inside the ear canal and are almost invisible as shown in Figure 1.9. Therefore they provide more comfort and cosmetic value. Because of the closer proximity to the eardrum and the resonance characteristic of the ear canal, less power is required to provide equal amounts of amplification compared to the other instrument types. The other hearing instruments also have the 'occlusion effect', which refers to the sensation of "talking in a barrel". This is greatly reduced or eliminated with CIC hearing aids. Another advantage is reduced wind noise [1].

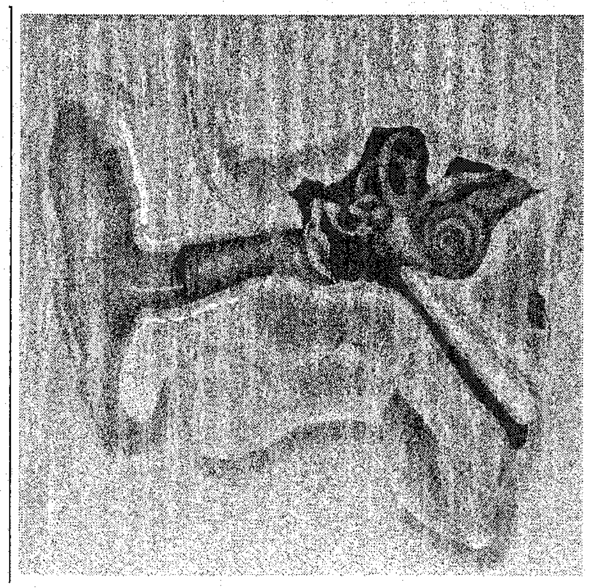


Figure 1.9 CIC Hearing instrument placement

1.5 Hearing Instrument Technologies

For persons with sensorineural hearing losses it is necessary that the frequency response is optimally fitted into their residual hearing area, and that the fitting is valid under various conditions, such as a change in input level. Hearing instruments ideally optimize speech intelligibility for the listener, while maintaining user comfort. In recent years, there has been a transition, in the design of hearing instruments, from analog technology to digital technology. In general, we can identify three major types of hearing instruments [1]:

Analog Hearing Instruments: This type of hearing instrument is also referred to as a conventional hearing instrument. The instrument comprises a microphone, an amplifier unit, a loudspeaker and a battery. The circuitry used in these hearing instruments is totally analog. There are several drawbacks to the use of analog technology, including difficulty in obtaining high resolution in the frequency domain; analog hearing aids usually have a maximum of two frequency channels. The instruments are adjusted manually through trimmer potentiometers; this adjustment is not accurate and can drift. These instruments generally apply only linear amplification.

Programmable Hearing Instruments: This type of hearing instruments is based on analog circuitry, where a memory module replaces the potentiometers. The memory module can be a RAM or EEPROM, which is accessed through an external microprocessor. They are basically analog hearing aids, whose control is improved through digital technology. Even if the patients hearing changes they can be reprogrammed instead of having to obtain a completely new instrument. Compression is achieved through sound-level dependent amplification.

Both analog and hybrid hearing aids, with only low order filtering possibilities, exhibit difficulties in obtaining appropriate corrective frequency responses.

Digital Hearing Instruments: These instruments convert the analog signal, received from the microphone, to a digital data stream, and process the signal with DSP (digital signal processing) algorithms. The output is converted again to an analog signal, which drives the loudspeaker. In the signal processing part, the sound is split into multichannels, where the frequency shaping of the sound signal is performed according to the patient's audiogram. Nonlinear amplification parameters are set once and the algorithm adjusts itself according to the intensity level of the input sound. Besides compression algorithms, speech enhancement algorithms are also used. These allows better overall sound quality and compensation of loudness growth. More sophisticated signal processing, such as adaptive noise cancellation and acoustical feedback cancellation, will be simplified by using digital processing.

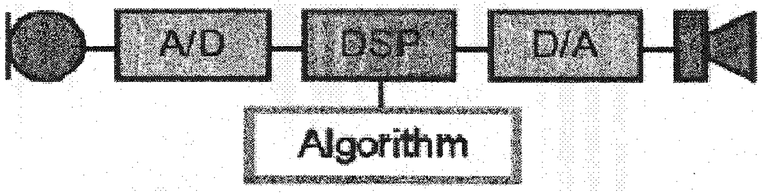


Figure 1.10 Digital hearing instrument structure [1]

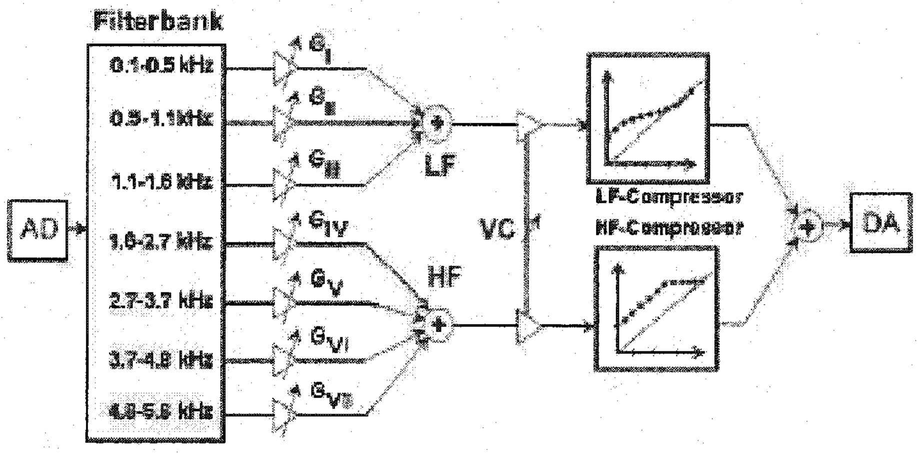


Figure 1.11 An example of the DSP part of the hearing aid block diagram

As shown in Figure 1.11, the DSP block contains a filterbank and compression blocks.

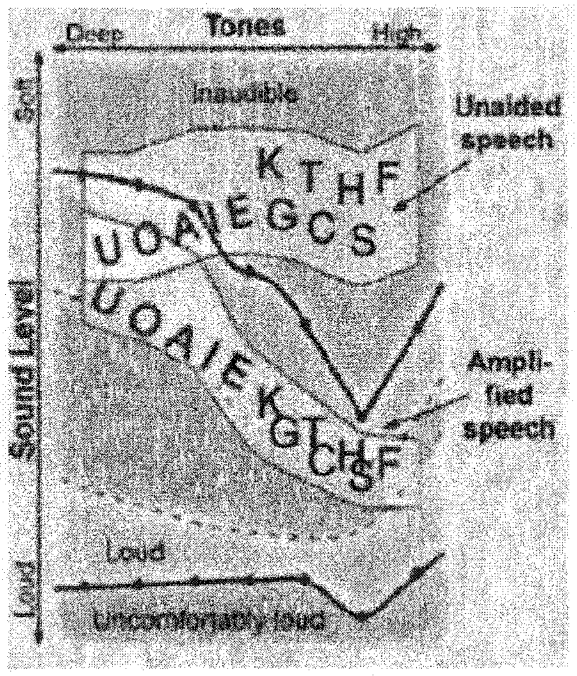
Filterbank

As plotted and tabulated in Section 1.3.2 sensorineural hearing loss can exhibit different types of frequency characteristics. The reason for this is based on the fact that the cochlea operates as a filterbank and damage to the organ of corti, sensing a specific band, results in narrowband hearing loss. Therefore, the hearing instrument should provide high frequency resolution in order to compensate properly. Considering that the bandwidth of a telephone line is between 200 Hz and 3.2 kHz, a frequency range up to 8 kHz would cover almost the entire speech components and most of the music components, as demonstrated in Figure 1.1. In terms of an octave scale, 90% of the sound information would be covered [1].

Digital filterbanks can be used to obtain an arbitrary magnitude response with exact linear phase. This provides considerable flexibility and ease whereby the characteristics of a digital filter can be instantaneously adjusted [13].

Compression

Because of the dynamic range reduction and rapid loudness growth, linear amplification can not be applied to hearing instrument sound signals. This would amplify soft signals to the audible range but loud signals would be amplified to the uncomfortable range. Therefore, modern hearing instruments apply compressive amplification, which maps the speech signal into the dynamic range of the hearing impaired person. This is demonstrated in Figure 1.12 [1].



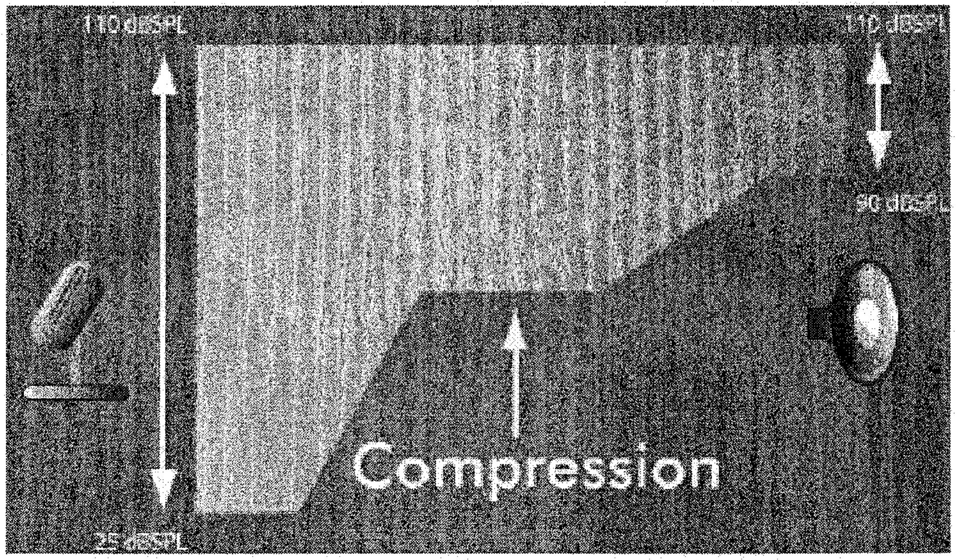


Figure 1.12 Compressive amplification

Low power and small size are two important constraints for a digital hearing aid. A hearing aid is required to run on a single cell zinc-air battery (V_{supply}= 1V) for a minimum of one week; thus, the maximum power consumption should be less than 1mA @ 1V = 1mW. The maximum dimensions for a CIC hearing are: length < 3mm, width <5mm, height < 2 mm [4]; a hearing aid integrated circuit should clearly fit into these dimensions, which constrains the sophistication of the processing circuitry, based on the fabrication technology.

The filterbank is a core part of current digital hearing aids and constitutes about half of the signal processing circuitry [14]. In this research we focus on the filterbank design and implementation.

1.6 MDLNS

As mentioned previously, the hearing loss compensation in a typical digital hearing instrument, is performed by separating the input signal into multiple frequency bands which are then compressed to allow the amplification of low level signals while maintaining the amplitude of high level signals. We thus require a processor that is able to both perform linear processing (band separation), and non-linear processing for signal compression. In order to be able to adequately represent the very low level signals, that are subject to the maximum amplification in the processor, very large word-lengths are required, and floating point representation is quite usual in this regard [11]. To be practically usable in a completely-in-canal (CIC) device, the digital circuitry needs to fulfill the joint requirements of low power consumption and small size. The Multidimensional Logarithmic Number System (MDLNS) is a recently developed number system [5] that appears to be a good candidate for implementing hearing instrument processors. Although the LNS has been previously considered for digital hearing aid processor [8], this research presents an exploration of the MDLNS for digital hearing aid circuitry. As with the LNS, the MDLNS provides a reduction in the size of the number representation, but the MDLNS promises a lower cost (area$\times$power) implementation of the arithmetic operations required in both the linear and non-linear domains of filtering and compression. In this research we apply the MDLNS to the construction of a finite impulse response (FIR) filterbank for digital hearing instruments.

We use the MDLNS representation for the filterbank design in order to both reduce the size of the number representation and to allow a relatively low cost implementation of the operations of multiplication. This number representation uses both a one and two digit MDLNS where the digits are represented by the indices of their bases. This provides a logarithmic type of representation with the added advantage of allowing orthogonal implementations of the index computation on each base.

In the MDLNS domain (as with logarithmic arithmetic with a radix of 2) we can more accurately represent smaller values compared to an integer binary representation, and the percentage error normalized to the signal magnitude remains quite constant. Figure 1.13 shows the signal-to-quantization noise ratio vs. input level of an input signal quantized to a single-digit 8-bit MDLNS and an equivalent 16-bit binary representation. The MDLNS representation provides a constant S/N ratio throughout the range of inputs, while the binary S/N ratio drops with input level. The MDLNS has a superior S/N ratio below 60 dB, the sound level of an average conversation;

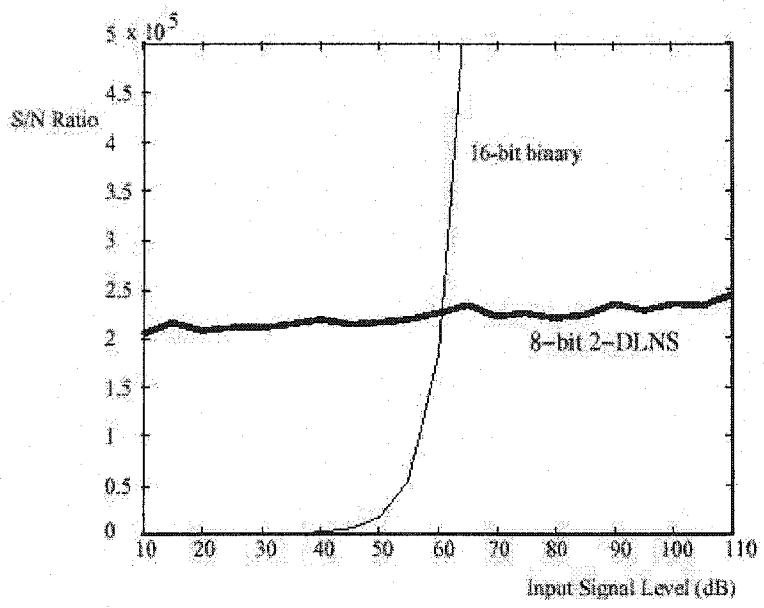


Figure 1.13 Signal-to-quantization noise ratio vs. input level

Figure 1.14 shows the distribution of filter coefficients from a study of 200 different designs using 9-bit quantization on the coefficients [5]. The distribution clearly shows that the density of filter coefficients is a Gaussian-like function centered on zero. Such a coefficient distribution is better represented by a logarithmic-like number system, such as the MDLNS, rather than a classical positional number representation (such as binary) [5]. So both input data and coefficients of digital filterbank are better represented by MDLNS than by a fixed-point binary representation.

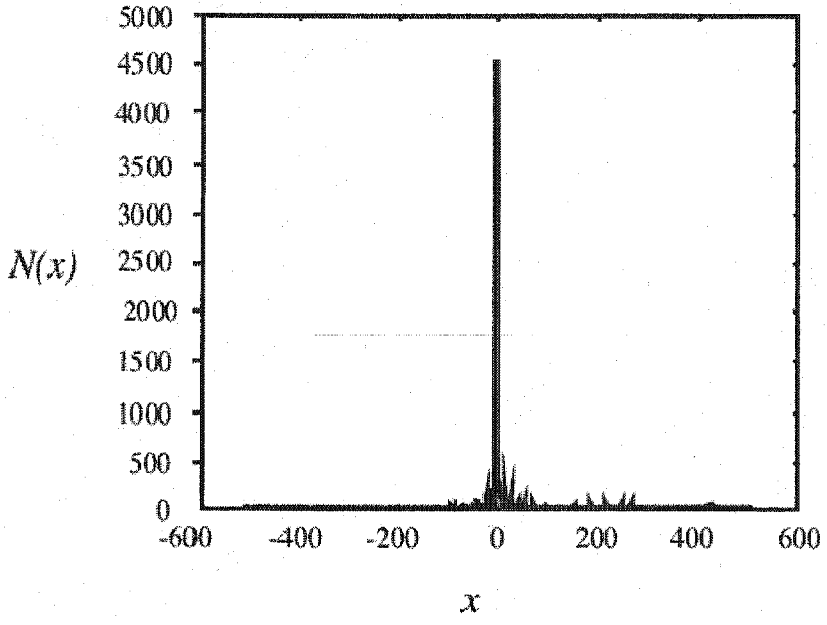


Figure 1.14 Filter coefficient distribution

1.7 Thesis Organization

This thesis is organized into seven chapters and four appendices. The first chapter (this chapter) is an introduction. Chapter 2 covers MDLNS background theory both in terms of the representation and the arithmetic. Chapter 2 also provides a comparison between the MDLNS and LNS number systems. Chapter 3 discusses current digital filterbank algorithms and a new MDLNS filterbank algorithm. Chapter 4 presents architectures of three filterbanks. Chapter 5 provides verification of the MDLNS filterbank architecture through the simulation of a microchip of an 8-band filter at a hardware description language level. Chapter 6 provides some comparisons between the new designs and recently published designs; the comparisons include noise, area and power dissipation. Chapter 7 presents conclusions and potential future work.

Chapter 2

MDLNS Background Theory

2.1 Introduction

The logarithmic number system (LNS) is an alternative to the classical binary representation and it has been the subject of some investigation [5], particularly in the field of digital signal processing (DSP), where the computation of inner (dot) products is a major computational step. In the LNS, multiplication and division are easy operations, whereas addition and subtraction are difficult operations, traditionally implemented by using large ROM arrays. Inner products computed in DSP algorithms are often between a predetermined set of coefficients (e.g., FIR filters or discrete transform basis functions) and integer data. For fixed point binary implementations, the uniform quantization properties (a constant error bound over all mapped input values) are perfectly matched to the mapping of most input data (the mapping of input data for non-linear hearing instrument processing is a counter-example), but often the predetermined coefficients are better suited to a non-uniform quantization mapping. A study of a large number of filter designs reveals a histogram that benefits from the quantization associated with a logarithmic mapping [5]. A logarithmic-like representation, multi-dimensional logarithmic number system (MDLNS), was recently introduced [6] that promises implementation improvements over the LNS while maintaining a logarithmic quantization distribution [5].

2.2 Representation

The MDLNS representation of a number differs somewhat from the traditional fixed radix form of representation. In a fixed radix positional system, a number is represented in the form:

$$\chi = \sum_{i=0}^{N} m_i r^i \tag{2.1}$$

where $m \in \{0, 1, \ldots, r-1\}$, i is an integer and r is the radix. For example in the decimal system $r = 10$, and in the binary system $r = 2$.

In the Logarithmic Number System (LNS)a number is represented by:

$$x = s_i 2^{a_i} \tag{2.2}$$

where a_i is an arbitrary real number.

We can build upon some well-established results on s-integers before defining a multi-dimensional logarithmic number system [5]. We start with two basic definitions.

Definition 1: An s-integer is a number whose largest prime factor does not exceed the s-th prime number.

For example, non-negative powers of two are 1-integers, numbers of the form $2^a 3^b$, a, b - non-negative integers, are 2-integers and so on.

Definition 2: Modified 2-integers are numbers of the form $2^a p^b$, p - odd prime.

Note that we do not impose restrictions on the sign of a and b in Definition 2.

Definition 3: A representation of the real number, x, in the form:

$$x = \sum_{i=1}^{n} s_i \prod_{j=1}^{b} p_i^{e_j^{(i)}} \tag{2.3}$$

where $s_i \in \{-1, 0, 1\}$ and $p_j, e_j^{(i)}$ are integers, is called a multidimensional n-digit logarithmic (MDLNS) representation of x. b is the number of bases used (at least two, the first one, that is, p_1, will always be assumed to be 2. The next two definitions are special cases of Definition 3.

Definition 4: An approximation of a real number x as a signed modified 2-integer $s2^a p^b$, is called a two-dimensional logarithmic representation of x.

Definition 5: An approximation of a real number x as a sum of signed modified 2-integers $\sum_{i=1}^{n} s_i 2^{a_i} p^{b_i}$ is called an n-digit two-dimensional logarithmic representation of x (2DLNS). The binary number system to be a subset of MDLNS if $b_i = 0$ for all i. If p=3 then x= $\sum_{i=1}^{n} s_i 2^{a_i} 3^{b_i}$. In this form the number is represented by the sign and the exponents. The exponent on the base of 2 is called the binary exponent, and the exponent on the base of 3 is called the ternary exponent. we will use a triple $\{s_i, b_i, t_i\}$ to represent x. If we fix the binary and ternary exponent to a signed, fixed length representation, we can represent any real number to arbitrary precision. We use this kind of 2DLNS representation as an example to illustrate some characters of MDLNS.

For single-digit representation of the form:

$$y = s2^b3^t \tag{2.4}$$

where $s_i \in \{-1, 0, 1\}$ and b, t are signed integers. In this case we have the following theorem [9].

Theorem 1: For every $\varepsilon > 0$ and every non-negative real number x, there exist a pair of integers a and b, such that the following inequality holds:

$$|x - 2^a3^b| < \varepsilon \tag{2.5}$$

We may therefore approximate, to arbitrary precision, every real number with the triple $\{s, b, t\}$. We may look at this representation as a two-dimensional generalization of the binary logarithmic number representation. The important advantage of this generalization is that the binary and ternary exponents are operated on independently from each other, with an attendant reduction in complexity of the implementation hardware. As an example, a VLSI architecture for inner product computation with the MDLNS proposed in [9][7], has an area complexity dependent entirely on the dynamic range of the ternary exponents. Providing that the range of the ternary exponent is smaller than the LNS dynamic range for equivalent precision, then we have the potential for a large reduction in the MDLNS hardware compared to that required by the LNS. We can capitalize on this potential improvement by placing design constraints on the ternary exponent size. For example, if we want to represent digital filter coefficients in the MDLNS, then we can design the coefficients in such a way that the ternary exponent is minimized; an integer programming task [5]. Although this approach is sound, and can produce modest improvements, generalizing the representation to multi-dimensions and/or multiple digits has the potential to bring about very large reductions in hardware complexity of DSP implementations.

It is important to note that an extension of the classical LNS to a multi-digit (or multi-component) representation does not provide any inherent advantages in terms of complexity reduction. We can show this for the special 2-digit 2-dimensional case with the following two theorems.

Theorem 2: Let x be an integer with the following 2-digit 2-base logarithmic approximation $x \approx s_1 2^{a1} p^{b1} + s_2 2^{a2} p^{b2}$. Then for $\left|x - (s_1 2^{a1} p^{b1} + s_2 2^{a2} p^{b2})\right| < 0.5$, the dynamic range of a_1, b_1, a_2, b_2 is $0.5 \cdot \log(x) + o(\log x))$.

Theorem 3: Let x be an integer with a 2-component LNS representation

$$x \approx s_1 2^{l_1} + s_2 2^{l_2} \quad . \tag{2.6}$$

Then for $\left|x - (s_1 2^{l_1} + s_2 2^{l_2})\right| < 0.5$ the dynamic range of l_1, l_2 is $\log(x) + o(\log x))$.

Our conjecture is that both theorems are generalizable to n-digits. We can see from Theorems 2 and 3, that the dynamic range of the exponents is reduced by a factor of 2 for the 2DLNS, but for the 2-component LNS system there is no reduction at all. Since we will demonstrate that hardware complexity for the MDLNS is exponentially dependent on the size of the second base(s) exponent, we clearly have a potential for quite considerable hardware reduction providing that the following inequality is met: $max(|b_1|, |b_2|) < max(|l_1|, |l_2|)$ [5].

2.3 Input Data Mapping

We will start this section using an example of a linear representation using the Double-Base Number System [9], DBNS, from which the MDLNS was developed.

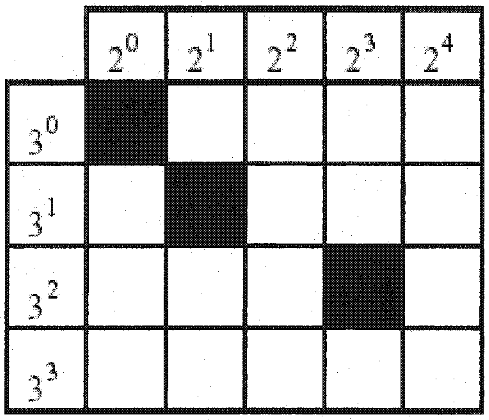


Figure 2.1 A 2-digit DBNS representation of 79

Figure 2.1 shows one of the possible mappings of 79 into the DBNS. The black squares represent the location of a digit. The digits used are $2^0 3^0 + 2^1 3^1 + 2^3 3^2 = 1+6+72 = 79$. The white squares represent the '0' digits and the black squares represent the '1' digits. From Figure 2.1 the DBNS is clearly a very sparse representation.

If negative exponents are used, the representation of real numbers, with arbitrary precision, can be obtained with single digits; in general, however, we can use more than one digit. Figure 2.2 shows the representation of the real number 7.25 using two digits $2^{-2}3^3 + 2^{-1}3^0 = 63.75 + 0.5 = 7.25$, and an expanded table using signed indices.

From our earlier definition of an MDLNS representation, we see that Figure 2.2 represents 7.25 using a 2-digit 2DLNS representation (1,-2,3), (1,-1,0) with bases 2 and 3. Note our short-hand for the digit as triples (*sign, binary exponent, ternary exponent*).

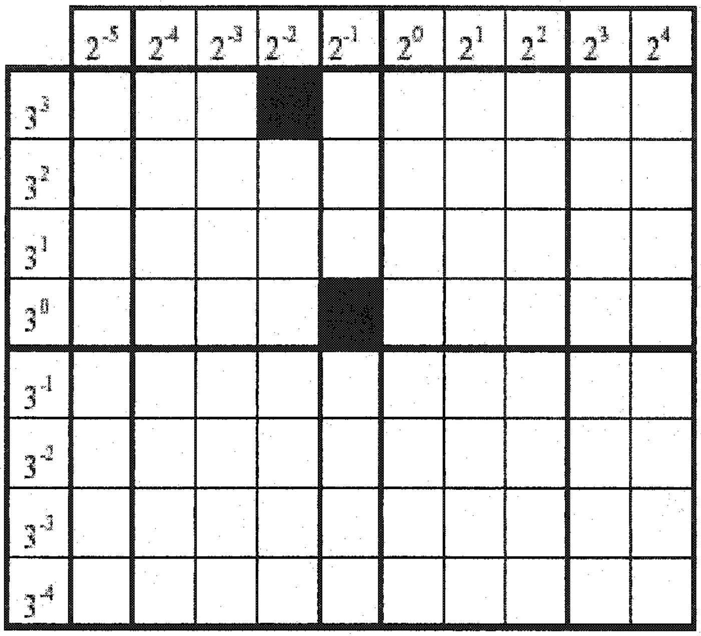


Figure 2.2 A 2DLNS representation of 7.25

Clearly only a small number of digits are required to represent a number in the 2DLNS. For example, the number 103 can represented, approximately, by the two 2DLNS digits (1,-24,18), (1,19,-8) and 2315 can be represented, exactly, by the two digits (1,0,7), (1,7,0). Since the MDLNS is a redundant number system, most numbers will have more than one representation. For example 4.25 has 3 error free two digit MDLNS representations, while 7.25 has a single unique, error free, two digit MDLNS representation. It is provable that every real MDLNS representations, but realistically the majority will have at most 4 or 5 error free representations. This redundancy can be useful in allowing you to choose the best possible representation for your application [10].

2.3.1 Error-Free Representations

As stated in the introduction, most often in DSP applications the input data has to be converted from analog to a fixed point binary value with a uniform quantization error bound. Mapping to integers has a quantization error bounded by 0.5 for all converted values. For a classical LNS representation (and also a 1-digit MDLNS representation) we do not have this uniform quantization accuracy so we have to choose a sufficient number of bits so that we will be able to maintain this conversion accuracy for the larger data values. In the multi-digit MDLNS we can mitigate this quantization problem; in fact, we can find certain MDLNS representations that are completely error free! [5]

Consider the case of the two odd prime bases, (3, 5).

A representation of a real number into forms given in Definition 3 to 5 is called error-free if there is zero approximation error. The next three theorems and one conjecture have provided new results about the error-free two-dimensional logarithmic representation of numbers [5].

Theorem 4: Every real number, x, may have at most 91 different error-free 2-digit two-dimensional logarithmic representations.

Theorem 5: The smallest positive integer with no error-free 2-digit two-dimensional LNS representation in the case of odd base three is 103.

Theorem 6: The smaller positive integer with no error-free 2-digit two-dimensional logarithmic representation in the case of odd base five is 43.

The following conjecture is based on extensive numerical calculations.

Conjecture 1: The smallest positive integer with no error-free 3-digit two-dimensional logarithmic representation in the case of odd base three is 4985.

It is important to note that such results will be available (and different) for every particular set of bases that we choose. In this case (that is, a 3-digit two-dimensional logarithmic representation with odd base three) we see that a 12-bit error-free mapping is available; a useful dynamic range for many DSP applications.

2.3.2 Non Error-Free Representations

Clearly, error-free representations are special cases of the MDLNS, but the extra degree of freedom provided by the use of multiple digits can mitigate the non-uniform quantization properties of the classical LNS [5].

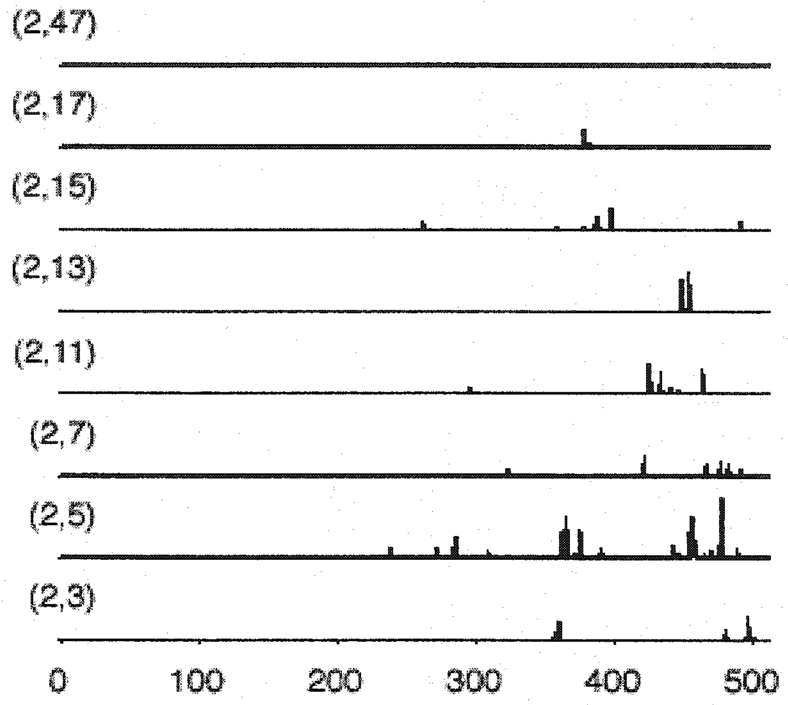


Figure 2.3 Absolute error (>0.5) for different second bases

To illustrate this, we present numerical results for mapping 10-bit signed binary input data to the 2-digit 2-D LNS where we treat the second base as a parameter. In order to demonstrate the ability to closely match input data with very small exponents, we have restricted the second base exponent to 3-bits. We are allowing the binary exponent to be somewhat larger, namely 6-bits, but, as we will see in the next section, this has very little bearing on the overall complexity of the inner product implementation (i.e., the hardware complexity is mainly driven by the dynamic range of the second exponents). As stated above, we require quantization errors to be <0.5 in order to match a binary representation. Figure 2.3 shows the results for parameters in the set {3,5,7,11,13,15,17,47}. The scale for each graph is between 0.5 and 1 for the absolute error. There are two observations: 1) there are very few values in any of the results where the error exceeds 0.5; 2) there appears to be no correlation between the density of errors >0.5 and the value of the second base.

For a second-base of 47 we find no errors that exceed 0.5, whereas for *integer* bases below this value we find some error. Of course, we could speculate on the use of real numbers rather than integers for the second base, and we have some recent evidence that we can indeed reduce the absolute error below 0.5 for small real numbers in the some range.

To compare these results with an implementation using a classical LNS representation, we need to determine the number of bits of the logarithm to produce an absolute error of <0.5. A previous study has found that we require n+3 bits for the logarithm in order to achieve this accuracy for an n-bit positive number [12]. If we assume that the hardware complexity of the classical LNS representation is driven by the number of bits in the logarithm, then we can see a potential for enormous reduction in the implementation complexity of the 2-digit 2-D LNS versus the classical LNS [5].

2.3.3 Mathematical Operations

An 2DLNS representation provides a triple, $\{s_i, b_i, t_i\}$, for each digit, where s_i is the sign bit and b_i, t_i are the exponents of the binary and non-binary bases. Thus a number, x, can be represented as given in Eqn. (2.7).

$$x = \sum_{i=1}^{n} s_i 2^{b_i} 3^{t_i} \tag{2.7}$$

Multiplication and Division

MDLNS multiplication and division are the simplest of the arithmetic operations. The equations for multiplication and division, given a single digit 2DLNS representation of $x = (s_x, b_x, t_x)$ and $y = (s_y, b_y, t_y)$, are [9]:

$$x \cdot y = (s_x s_y, b_x + b_y, t_x + t_y) \tag{2.8}$$

$$x/y = (s_x s_y, b_x - b_y, t_x - t_y) \tag{2.9}$$

Eqn. (2.8) and Eqn. (2.9) show that single digit 2DLNS multiplication can be implemented in hardware using two independent binary adders and simple logic for the sign correction. As we start to add digits to the representation we will face the equivalent of implementing multiplication with the addition of partial products. A 2-digit representation will produce four independent partial products that will have to be added, and since addition is an expensive operation we have to deal with this in a special way. In this thesis we will use a multiplier accumulator structure that eliminates much of the complexity of performing addition within the 2DLNS representation. Multi-digit arithmetic is briefly dealt with later in this section.

Addition and Subtraction

Unfortunately, as with logarithms, addition and subtraction operations are not as simple as multiplication and division operations. Addition and subtraction must be handled through a set of identities and look-up tables. The identities are [9]:

$$\begin{aligned} 2^a 3^b + 2^c 3^d &= 2^a 3^b (1 + 2^{c-a} 3^{d-b}) \\ &\approx 2^a 3^b \Phi(c-a, d-b) \end{aligned} \tag{2.10}$$

$$\begin{aligned} 2^a 3^b - 2^c 3^d &= 2^a 3^b (1 - 2^{c-a} 3^{d-b}) \\ &\approx 2^a 3^b \Psi(c-a, d-b) \end{aligned} \tag{2.11}$$

The operators Φ and Ψ are lookup tables that store the precomputed values of:

$$\Phi(x, y) = 1 + 2^x 3^y \approx 2^\alpha 3^\beta \tag{2.12}$$

$$\Psi(x, y) = 1 - 2^x 3^y \approx 2^\gamma 3^\delta \tag{2.13}$$

The use of large look-up tables, implemented through the use of ROMs, for the evaluation of addition and subtraction operations is the traditional approach in systems such as the

Logarithmic Number System [10]. This technique is only feasible for very small ranges of 2DLNS numbers. It is more practical to convert the 2DLNS numbers to binary and perform the addition and subtraction using binary representation.

The conversions from 2DLNS to binary will still require a look-up table, but one that is much smaller than required for handling 2DLNS addition and subtraction. The look-up table is used to convert the second base portion of the 2DLNS number into a binary representation. Therefore the size of the look-up table is dependent on the number of bits used to represent the second base exponent.

Multi-digit MDLNS Arithmetic

Multi-digit MDLNS arithmetic is simply an extension of the single digit MDLNS arithmetic, and is necessary when numbers are represented by more than one MDLNS digit. When performing a computation using multi-digit MDLNS each digit can be treated as an independent MDLNS number and the operations handled separately. For example, if X and Y are 2 digit MDLNS numbers such that $X = x_1+x_2$ and $Y = x_1+x_2$ then:

$$X \cdot Y = (x_1 + x_2)(y_1 + y_2) = x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2 \qquad (2.14)$$

where x_i and y_i are single digit MDLNS numbers.

Eqn. (2.15) is an example of 2 digit MDLNS multiplication of 27 and 23

$$\begin{aligned}
&(2^2 3^2 - 2^0 3^2) \cdot (2^5 3^0 - 2^0 3^2) \\
&= 2^2 3^2 \cdot 2^5 3^0 - 2^2 3^2 \cdot 2^0 3^2 - 2^0 3^2 \cdot 2^5 3^0 + 2^0 3^2 \cdot 2^0 3^2 \\
&= 2^{2+5} 3^{2+0} - 2^{2+0} 3^{2+2} - 2^{0+5} 3^{2+0} + 2^{0+0} 3^{2+2} \\
&= 2^7 3^2 - 2^2 3^4 - 2^5 3^2 + 2^0 3^4 \\
&= (1152)_{10} - (324)_{10} - (288)_{10} + (81)_{10} \\
&= (621)_{10}
\end{aligned} \qquad (2.15)$$

The independence of the arithmetic operations is very important, as it allows for parallel architectures. Furthermore, these parallel structures can be asynchronously clocked, reducing the power requirement of the circuit [10].

2.4 Hardware Complexity

In order to provide complexity results for the MDLNS inner product computation unit, we expand on the inner product processor architecture initially developed for the 1-Digit 2-D LNS [9]. The processor can be used in a filterbank for 1-D convolution.

2.4.1 Single Digit Computational Unit

Figure 2.4 shows the structure of the proposed single-digit computation unit (CU). Since we do not require to retain the MDLNS representation of the accumulated output, and also since the CU is feedforward, we can use the MDLNS domain for the coefficient multiplication and a binary representation for the accumulated output.

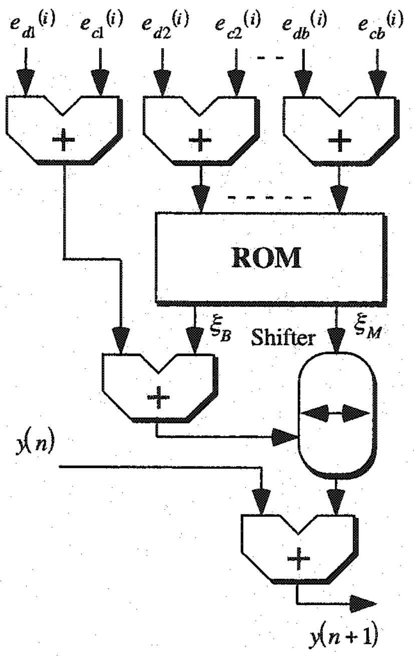


Figure 2.4 1-digit MDLNS inner product computation unit

The multiplication is performed by parallel small adders for each of the data and coefficient base exponents. The addition output for each of the b-1 odd bases is concatenated into an address for a lookup table (ROM). This table produces an equivalent floating point value for the product of the odd bases raised to the exponent sum, as shown below:

$$\prod_{j=2}^{b} p_j^{(e_{dj}^{(i)} + e_{cj}^{(i)})} \approx 2^{\xi_B} \cdot \xi_M \tag{2.16}$$

We note that since the size of the exponents of each odd base in an MDLNS representation (where there are at least 2-digits and 2 bases) can be very small (<4 bits), then the maximum address input to the ROM is given by $4(b-1)$ bits. This is an 8-bit address table for a 3-D LNS. The shifter in the floating point to binary (2's complement) conversion part of the unit also handles the sign of the product (not shown in Figure 2.4 for brevity).

For large dimensional LNS, we can also consider the use of unity approximants to reduce the output of each odd-base adder to the number of bits of the input exponents (or even less if we are willing to accept the increased mapping error). This reduction process stores a small number of unity approximants that can be added in parallel to the output of the odd-base adders. The reduced input to the ROM is selected from these parallel results. The ROM input address size is now reduced by (b-1) bits. This is a subject of current research work and will not be explored further here. For the complexity analysis in Section 2.4.3., we will assume the structure of Figure 2.4 [5].

2.4.2 n-digit Computational Unit

The n-digit computational unit is a simple parallel extension of the 1-digit unit. Each of the units computes the binary output for one of the digit combinations. As an example, consider multiplying an accumulating sequence, y, with a coefficient, x, $z = x \cdot y$, where:

$$y = \sum_{i=1}^{2} s_i^{[y]} \prod_{j=1}^{2} p_j^{e_j^{[y](i)}} ; \qquad x = \sum_{i=1}^{2} s_i^{[x]} \prod_{j=1}^{2} p_j^{e_j^{[x](i)}}$$

We can perform this with 4 parallel 1-digit units, where the (u, v) unit computes:

$$z_{u,v} = s_u^{[y]} \cdot s_v^{[x]} \prod_{j=1}^{2} p_j^{(e_j^{[y](u)} + e_j^{[x](v)})} \tag{2.17}$$

Clearly there are n^2 such units in an n-digit MDLNS. The parallel outputs are summed using an adder tree [5].

2.4.3 MDLNS Complexity Comparison

In terms of complexity comparison with the classical LNS, we need to compare each system based on the same quantization properties.

This will require a knowledge of the change in the size of the odd-base exponents as a function of dimensionality and number of digits. At the moment this appears to be a rather intractable task and so we have to resort to exhaustive search analysis. Here we report the results from 10 and 12-bit unsigned binary dynamic range (these are taken from typical video filter requirements). From Section 2.3.2., we obtain the number of bits for the classical LNS as n +3 shows these results.

Our assumption in Table 2.1 is that the ROM size dominates the hardware complexity. For the classical LNS and 1-digit 2D LNS, this is a valid assumption. For the 2-digit LNS the ROM size is very small and the other components in the architecture will undoubtedly be important in the overall complexity. Even so, it is clear that there is a very substantial reduction in the hardware complexity for the 2-digit case. We are currently performing a comparison analysis for a much larger set of digits and dimensions. Our initial findings point to the number of digits as being the primary contributor to the hardware savings [5].

Table 2.1Classical LNS and 2D LNS Comparison

Binary Dynamic Range	LNS	1-digit 2D LNS	2-digit 2D LNS
10-bits	8K	1K(10bit odd-base exponent)	64 words (4-bit ternary expo-nent)
12-bits	32K	4K (12-bit odd-base exponent)	128 words (5-bit ternary exponent)

Chapter 3

Digital Filterbank Algorithms

3.1 Introduction

There are five major forms of digital filterbanks that we compare here: QMF, Wavelet, Interpolated FIR (IFIR), DFT modulated, Parallel FIR. The first two banks are rejected based on the fact that the magnitude responses of the extended QMF bank and wavelet filter bank do not meet our frequency resolution criteria for hearing aids. The wavelet filter bank also does not allow perfect reconstruction [1].

This chapter covers the remaining three forms of digital hearing aid filterbank algorithm: Interpolated FIR (IFIR) filterbank, DFT modulated filterbank and parallel FIR MDLNS filterbank.

3.2 Design Constraints

The performance of a filterbank determines the frequency resolution of the instrument and the gain limits in each frequency band. The constraints listed at the end of the first chapter for the hearing instrument design are also the constraints for the digital filterbank design. The impacts of these constraints on digital filterbank design are explained in this section.

Frequency Range: It is shown in the first chapter that the frequency range of human hearing is from 20 Hz to 20 kHz. Because of the octave band characteristic of human hearing, good quality sound can still be achieved with half the frequency range coverage. In filter bank designs, 16 kHz is taken as sampling frequency. Because of the fact that spectrum coverage of a system is half of its sampling frequency, up to 8 kHz of the input sound signal is processed. This will cover the first nine octaves in Table 1.2 [1]

Number of Channels: Another important constraint is the frequency resolution. The monitoring of hearing loss is done through the generation of audiograms, which record measurements at eight different frequencies. Therefore 8 channels is an acceptable resolution for hearing instruments with more resolution at lower frequencies because of the octave characteristic of human hearing [1].

Stopband Attenuation: The stopband attenuation in each channel determines the gain range of the hearing instrument, and at least 50 dB of gain adjustment in each bank are required. The order of the filter is proportional to stopband and passband attenuation. When the order of the filter increases, the group delay and implementation cost increases. Therefore the tradeoff between these parameters should be well adjusted to achieve an optimum design [1].

linear phase: In a compression system, gain changes are dynamic. This may cause anomalies in the overall frequency response if phase differences exist between adjacent bands. To avoid these undesirable frequency response notches or peaks at the band edges (which frequently occur in analog systems), it is necessary to constrain the filter channel impulse responses to be linear phase and of equal delay.

Power Consumption: The power consumption is one of the most important constraints in filter bank design for hearing instruments. To achieve long battery life the circuitry inside the instrument should consume minimum power.

The most power consuming arithmetic unit in a VLSI implementation is multiplication [1] and the number of multiplications should be kept as small as possible. Besides the multiplication per unit time (MPU), addition per unit time (APU) is another important parameter to estimate the power consumption of the filterbank design [1].

Size: The research on filterbank design is for CIC model hearing instruments. Therefore the size of the filterbank structure should be kept as small as possible. This implies the use of minimum number of digital elements in filterbank structure, which also effects the power consumption.

3.3 IFIR Filterbank Algorithm Design

3.3.1 Introduction

This filterbank employs linear phase interpolated FIR filters and their complement. Properties of these filters and filter bank design procedure are explained in this section.

This technique enables the implementation of computationally efficient FIR filters. Instead of designing a very high order FIR filter, which satisfies the desired ripple and cutoff frequency specifications, the frequency response is achieved by cascading two or more FIR filters [13]. The basic idea is to implement the filter as a cascade of two FIR sections, where one section generates the sparse set of impulse response values with every L^{th} sample being nonzero, and the other section performs the unwanted passband suppression. These two sections can be defined as prefilter and image suppressor. The technique is called the interpolated FIR, because of the fact that the image processor block is applying an interpolation scheme to the impulse response samples of the prefilter. This interpolation extracts the desired narrow band filter.

3.3.2 Interpolated FIR filters

The approach with the IFIR technique is to construct a model filter of order N-1, for example a lowpass filter according to Figure 3.1a. The next step is to interpolate it with a factor L, that means that the impulse response $h(n)$ of the filter is filled with L-1 zeros between each filter coefficient with the new impulse response:

$$h_I = \begin{cases} h\left(\frac{n}{L}\right) & n = 0, L, 2L, \ldots \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

and with the z-transform:

$$H_I(z) = \sum_{m=-\infty}^{\infty} h_I(m)z^{-m} = \sum_{k=-\infty}^{\infty} h(k)z^{-kL} = H(z^L) \tag{3.2}$$

This can be interpreted as a repetition of images of the original lowpass function (Figure 3.1b). Thus, apart from the repetition, there is a narrowing of the transition bands.

3.3.3 Complementary halfband linear-phase FIR filters

It can be shown [18] that, with a lowpass halfband linear-phase FIR filter of even order $N-1$, with transfer function $H(z)$, and under certain symmetry constraints, it is possible to achieve the complementary half band highpass filter $H_c(z)$ as:

$$H_c(z) = z^{-(N-1)/2} - H(z) \tag{3.3}$$

The required symmetry constraint is that the frequency response exhibits symmetry about $\pi/2$ if the passband and stopband ripple are equal. The output of the filter can, according to eqn. (3.3), be obtained by:

$$y_c(n) = x(n-[N-1]/2) - y(n) \tag{3.4}$$

Thus, the complementary output will be reached by a single subtraction. Furthermore, it can be shown [18], that the filter coefficients are constrained to be:

$$h(n) = \begin{cases} 0 & n = odd \neq (N-1)/2 \\ 0.5 & n = (N-1)/2 \end{cases} \tag{3.5}$$

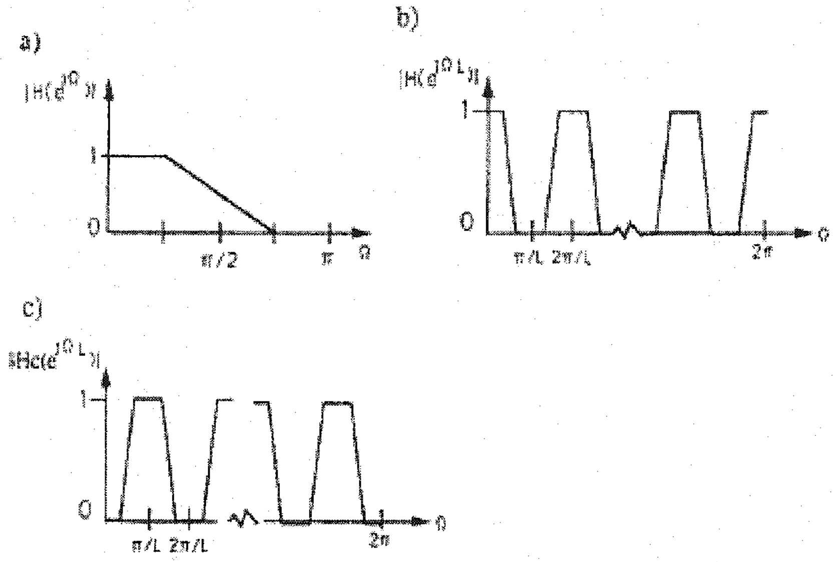


Figure 3.1 a) Lowpass modelfilter b) Interpolated model filter c) Complementary interpolated model filter

3.3.4 Prototype filter $H_1(z)$

Consider the magnitude response in Figure 3.1b, by applying all of the FIR properties mentioned above on that filter; i.e., an interpolated halfband linear-phase FIR filter. It is also possible to achieve the complementary filter as in Figure 3.1c. That filter can be realized according to Figure 3.2.

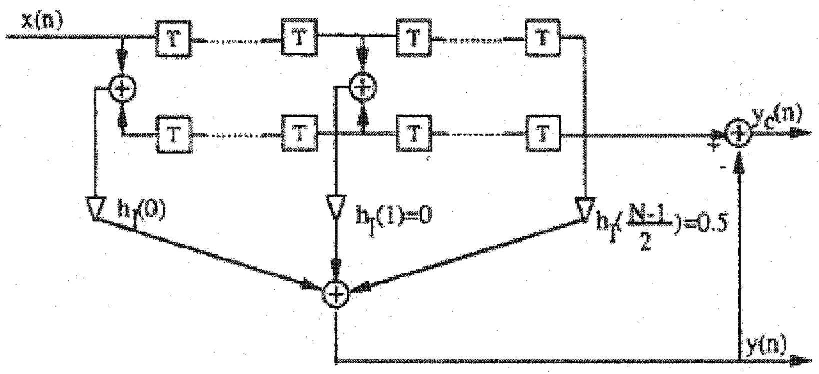


Figure 3.2 Realization of an interpolated complementary linear-phase FIR filter

We interpret that filter as a prototype filter $H_1(z)$ with each passband being a prototype band of a final single band output from the filterbank and interpret the complementary prototype filter $H_{1c}(z)$ in the same manner. These two filters define all bands included in the filterbank. The task for the rest of the filterbank structure (Figure 3.3) is to separate each individual passband to the outputs $[\hat{y}_1(n) - y_M(n)]$. The separation is carried out by two sub-filterbanks H_{S1} and H_{S2}.

Consider Figure 3.1b. By using an interpolation factor L=8 the prototype filter will contain five prototype bands. The complementary filter will contain 4 prototype bands.

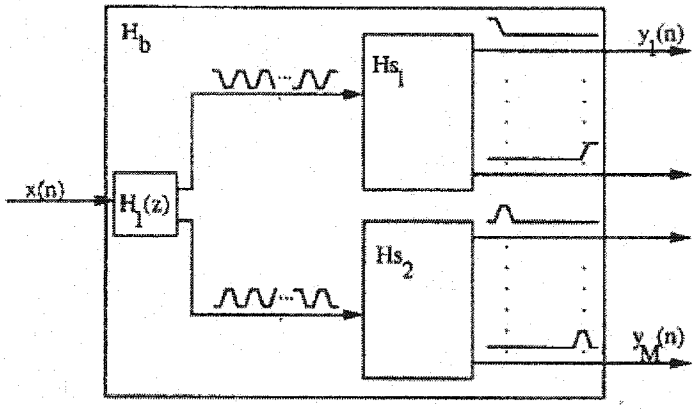


Figure 3.3 Filterbank structure with prototype filter and subfilterbank

$H_1(z)$	$h_1(0)=h_1(48)$, $h_1(16)=h_1(32)$, $h_1(24)=0.5$
$H_2(z)$	$h_2(0)=h_2(24)$, $h_2(8)=h_2(16)$, $h_2(12)=0.5$
$H_3(z)$	$h_3(0)=h_3(28)$, $h_3(4)=h_3(24)$, $h_3(8)=h_3(20)$, $h_3(12)=h_3(16)$, $h_3(14)=0.5$
$H_4(z)$	$h_4(0)=h_4(12)$, $h_4(4)=h_4(8)$, $h_4(6)=0.5$
$H_5(z)$	$h_5(0)=h_5(10)$, $h_5(2)=h_5(8)$, $h_5(4)=h_5(6)$, $h_5(5)=0.5$
$H_6(z)$	$h_6(0)=h_6(6)$, $h_6(2)=h_6(4)$, $h_6(5)=0.5$
$H_7(z)$	$h_7(0)=h_7(30)$, $h_7(6)=h_7(24)$, $h_7(12)=h_7(18)$, $h_7(15)=0.5$
$H_8(z)$	$h_8(0)=h_8(2)$, $h_8(1)=0.5$

Table 3.1IFIR filterbank coefficients not equal to zero

Prototype filter synthesis. The prototype filter $H_1(z)$ was synthesized by using the McCellan-Parks algorithm according to the specification above. The filter coefficients that have to be executed are listed in Table 3.1. Thus, the prototype filter and complementary filter can be implemented by 3 multiplications in accordance with Figure 3.2.

3.3.5 Sub-filterbanks Hs_1 and Hs_2

The separation of a single prototype band by the following filters will be facilitated since the demands on the transition bands can be lowered. The philosophy when designing the sub-filterbanks is to try to keep the number of multiplications down in the same manner as for the prototype filter by using special FIR properties.

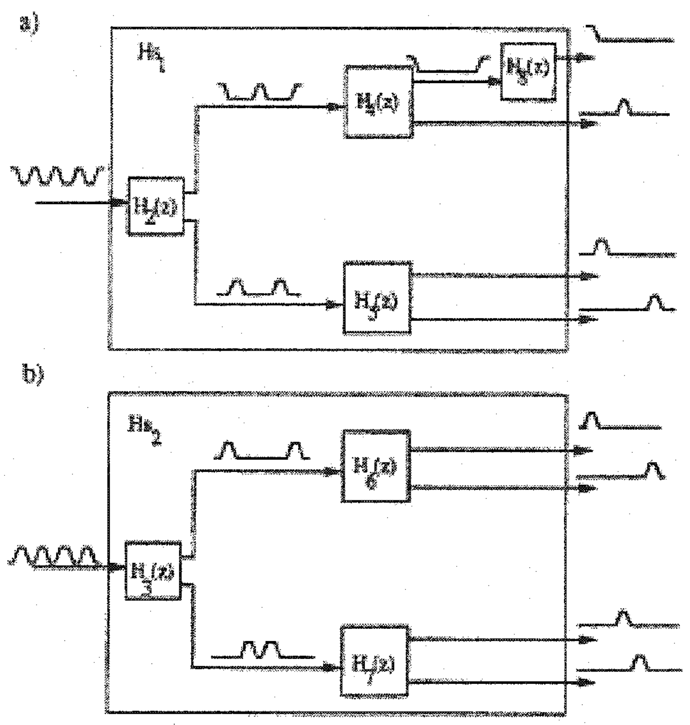


Figure 3.4 a) Sub-filterbank Hs_1 b) Sub-filterbank Hs_2

Consider Figure 3.4a. It is possible to use an interpolated half-band linear-phase FIR filter by applying a filter $H_2(z)$ on the sub-filterbank input signal, which has passband for every other passband of the prototype band of $H_2(z)$ will be hidden. The complementary filter $H_{2c}(z)$ then will contain those hidden bands. The output of $H_{2c}(z)$ is then processed by filter $H_4(z)$ which also separates every other prototype band. The two bands from $H_4(z)$ will be separated by filter $H_8(z)$. The complementary filter $H_{8c}(z)$ will not be used since that band is intended for antialising transition. The complementary output $H_{4c}(z)$ will contain a single prototype band and the signal path is completed. The same philosophy is also applied on the sub-filterbank in Figure 3.4b, where every filter is at least a complementary linear-phase FIR filter. By inspection of Table 3.1, the number of executable multiplications in total is found to be 27, if we consider that a multiplication with 0.5 is a simple shift the total sum is 19. The stop-band attenuations, for all the filters in the sub-filterbanks, are 40 dB.

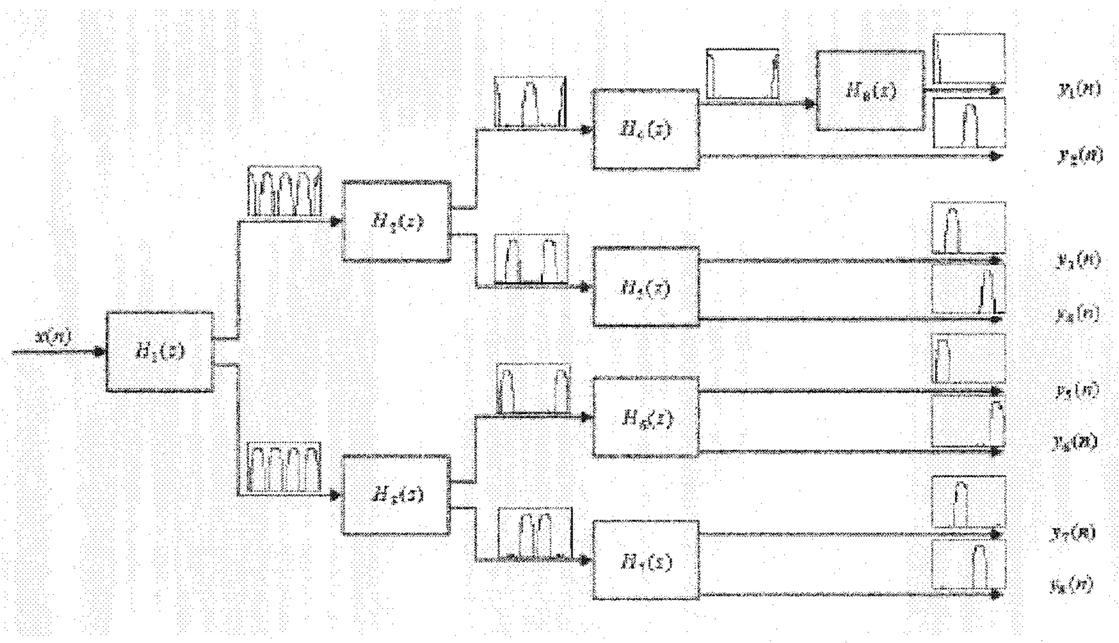


Figure 3.5 IFIR filterbank structure [1]

3.3.6 IFIR filterbank with 60 dB noise level

In Table 1.3 it is shown that a filterbank with 40-45 dB gain increase is needed to compensate moderate level hearing loss. Therefore the design constraints of the filters are increased to 60 dB stopband attenuation. The structure of the filter bank is the same. The only difference is the increase in the order of filters to achieve higher stopband and passband ripples. The passband ripples are between +0.02 dB and -0.02 dB. The noise floor is at 59 dB as shown in Figure 3.6. All pass bands of the filter bank are plotted together in Figure 3.7 because the effects of the individual stopband characteristics are observed best in the gain increase effect on the overall magnitude response. The ripples of this overall magnitude response are in the range of +0.01 dB to -0.01dB [1]. There is a large drop in composite magnitude response near 8KHz. The influence of 50 dB gain on the adjacent band is below 3 dB, which is an acceptable value. The buffer level between the noise level and gain should be again around 9 dB. Therefore gain up to 50 dB can be applied in this filter bank structure. It is enough to compensate moderate level hearing loss. The group delay is increased to 103 samples, which corresponds to a latency of 6.4

ms at a 16 kHz sampling frequency, and this is below the 10 ms maximum group delay constraint [1].

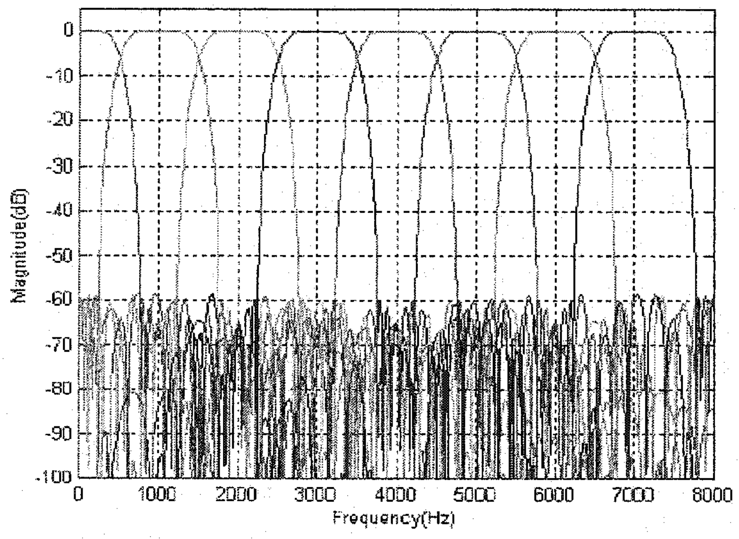


Figure 3.6 Magnitude response of 60 dB IFIR filterbank [1]

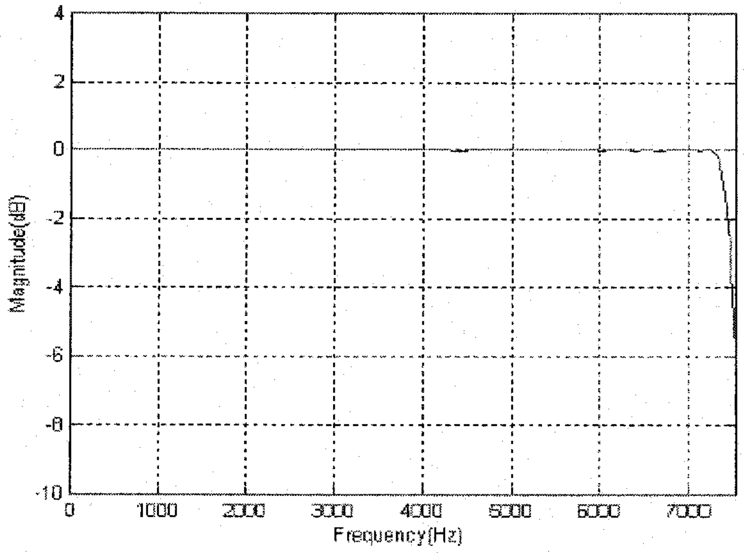


Figure 3.7 Maximally flat overall magnitude response of 60 dB IFIR filterbank [1]

3.4 DFT Filterbank Algorithm

The design uses an oversampled, weighted overlap-add (WOLA) DFT filterbank [23] to split the input signal into 16 frequency bands. Much greater computation efficiency is obtained when the filter response shapes are realized as a series of modulations of a low-pass prototype filter covering the entire frequency range. These modulations produce identical filter shapes resulting in a uniformly spaced filterbank.

A prototype filter is modulated with the aid of a DFT, and the signal is separated into bands with these modulated filters. The data rate is reduced with the aid of an analysis filter bank and the signal is recovered at its original rate through a synthesis filter bank. The details of this filterbank can be found in [15]

The output signal $X_k(m)$ for the k th channel of the filterbank analyzer can be expressed in the form

$$X_k(m) = \sum_{n=-\infty}^{\infty} h(mM-n)x(n)W_K^{-kn}, \quad k = 0, 1, \ldots, K-1 \tag{3.6}$$

$$W_K = e^{j\left(2\frac{\pi}{K}\right)} \tag{3.7}$$

$$W_K^{-kn} = e^{-j\left(2\pi k\frac{n}{K}\right)} \tag{3.8}$$

the input signal x(n) is modulated by the function $e^{j(2\pi/k)}$ and lowpass filtered by the filter $h(n)$. It is then reduced in the sampling rate by a factor M to produce the channel signal $X_k(m)$. The filter $h(n)$ in this system, the analysis filter, determines the width and frequency response of each of the channels.

The DFT filterbank synthesizer interpolates all the channel signals back to their high sampling rate and modulates them back to their original spectral locations. It then sums all the channel signals to produce a single output.

$$\hat{x}(n) = \sum_{-\infty}^{\infty} f(n-mM)\frac{1}{K}\sum_{k=0}^{K-1}\hat{X}_k(m)W_K^{kn} \tag{3.9}$$

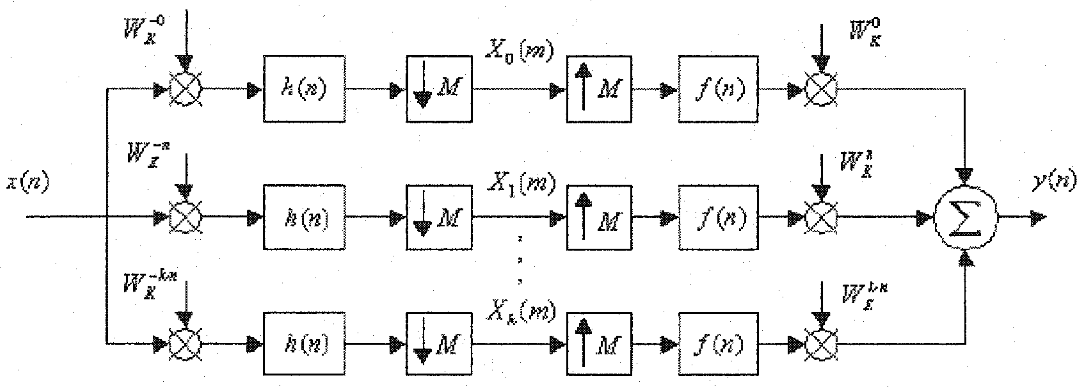


Figure 3.8 Complex modulator model of the DFT filterbank

Equations (3.6) and (3.9) form the mathematical basis for all the realizations of the DFT filterbank analyzer and synthesizers.

In general, the most concise alias-free representation is only obtained if the filter bands directly abut each other with complete frequency coverage and no overlap. Although this is impossible, developments have led to filterbanks with slightly overlapping bands designed in such a way that aliasing distortion generated in the analysis stage is exactly canceled by imaging distortion in the synthesis stage [22].

For hearing aid use, the frequency splitting is performed for the purpose of modifying the spectral shape of the input signal. Hearing aid fitting requires a wide gain adjustment range, and here the alias cancellation theory is invalid and critical sampling is insufficient.

This problem necessitated the development of an oversampled filterbank, which increases the data rate, and is the price that must be paid for gain adjustability without aliasing.

3.4.1 Oversampled WOLA Filterbank [22]

Simulations show that large band gain adjustments (>20dB) in a critically sampled filterbank result in severe aliasing distortion because aliasing cancellation conditions are violated. Distortion occurs because imaging in adjacent bands no longer cancels when the gain is adjusted. By over-sampling, the aliased images are placed further away from gain adjusted bands so that they can be rejected by the thesis filter, and it is necessary to oversample by at least a factor of two to reduce the level of uncanceled aliasing generated when band gains differ greatly. The selected design uses an oversampled, weighted overlap-add (WOLA) DFT filterbank [15][16][17] to split the input signal into 16 frequency bands. This filterbank uses modulation via the DFT to replicate a single prototype filter into 32 complex filter bands. This modulation produces identical filter shapes and results in a uniform filterbank. At a sampling frequency of 16 kHz, the resulting bands are 500 Hz wide. Total computational complexity is 46 multiply-accumulates per output point. Group delay is 12.5 ms.

To achieve sufficient frequency resolution at low frequencies, a uniform filterbank requires a larger number of bands than would be required by a non-uniform filterbank. Fortunately, the DFT method described above generates a large number of bands at low computational expense. This large number of bands is required to achieve a good fit to audiometric data at low frequencies (which is normally given on a log-frequency scale). As a result of the linear frequency spacing and large number of channels, the high frequency band spacing may be greater than necessary. Thus, it may be advantageous to group bands at high frequencies for gain adjustment purposes. It is important to realize that better frequency resolution is available only at the expense of greater signal delay for *any* filterbank method. Long delays are not desirable - delays of 6-8 ms are reported to be just noticeable. Delays longer than 20 ms may cause interference between speech and visual integration. Clearly, delays below 8ms are preferred.

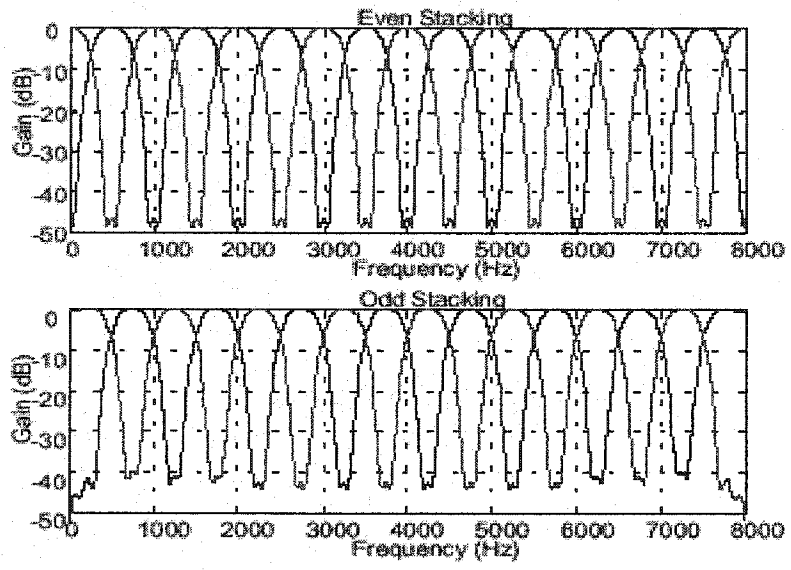


Figure 3.9 Frequency response of DFT filterbank channels for odd and even channel stacking arrangements (16 channels, sampling rate is 16kHz)

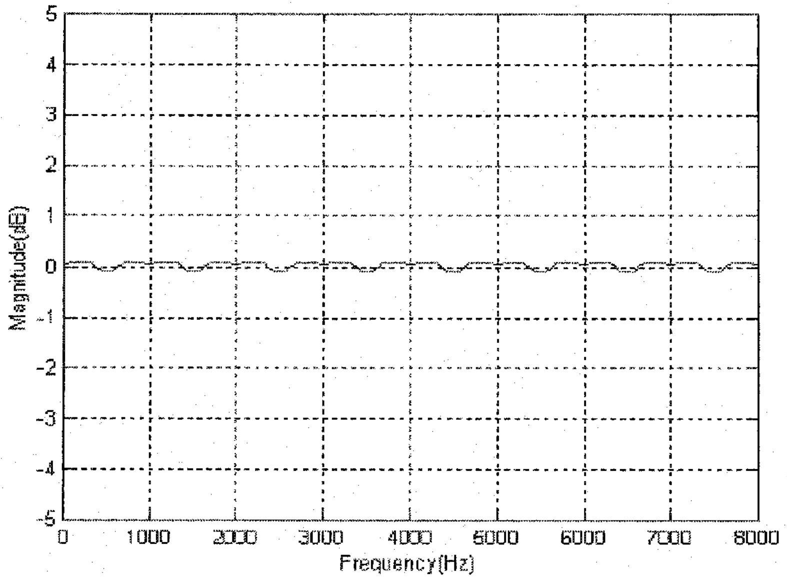


Figure 3.10 Overall magnitude response of the DFT filter bank [1]

The overall magnitude response is not flat. The ripple is ± 0.15 dB.

Both analysis and synthesis filters are linear phase, but since, in the filterbank structure, they are divided into their polyphase components, the linear phase property is partly lost [1].

3.4.2 Prototype Nyquist Eigenfilter

A combined analysis/synthesis filter can be used to reduce the memory requirements of the filterbank, where the synthesis filter is generated by decimating the analysis filter. For our experiments, the filter was designed using the equi-ripple Nyquist eigenfilter method, outlined in [19]. This method of designing linear-phase FIR filter minimizes a quadratic measure of the error in the passband and stopband. The method is based on the computation of an eigenvector of an appropriate real, symmetric, and positive-definite matrix. The proposed design procedure is general enough to incorporate both time and frequency-domain constraints.

The least-squares approach to approximate a design response, $D(\omega)$, with a type-1 linear-phase FIR filter transfer function $H(z)$ is to formulate an error function

$$E_{LS} = \int_R [D(\omega) - H_0(e^{jw})]^2 d\frac{\omega}{\pi} \tag{3.10}$$

where R is the region $0 \le \omega \le \pi$, but excluding the transition band. Where $H_0(e^{j\omega})$ is real-valued, given by

$$.H_0(e^{j\omega}) = \begin{matrix} \sum_{n=0}^{M} b_n \cos n\omega & N-1 \text{ even} \\ \sum_{n=0}^{M-1} b_n \cos\left(n + \frac{1}{2}\right)\omega & N-1 \text{ odd} \end{matrix} \tag{3.11}$$

The parameters b_n are found by minimizing $E_{LS.}$ The quantity M in eqn. (3.11) is defined as $M = \frac{(N-1)}{2}$ for even $N-1$, and $M = \frac{N}{2}$ for odd $N-1$. Defining

$$b = \begin{cases} [b_0 \quad b_1 \ldots b_{M-1} \quad b_M]^t & N-1 \quad even \\ [b_0 \quad b_1 \ldots b_{M-2} \quad b_{M-1}]^t & N-1 \quad odd \end{cases} \tag{3.12}$$

The total error of filter design need to be minimized is

$$E = b^t P b \tag{3.13}$$

A real, symmetric, positive-define matrix, **P**, can be constructed with a weighting variable α for passband and stopband error measure. The elements of **P** are given by:

$$P(n,m) = \begin{matrix} \frac{(1-\alpha)}{\pi}\int_0^{\omega_p}(1-\cos n\omega)(1-\cos m\omega)d\omega \\ +\frac{\alpha}{\pi}\int_{\omega_s}^{\pi}(\cos n\omega \cos m\omega)d\omega \end{matrix} \quad , N \text{ odd} \tag{3.14}$$

$$P(n,m) = \begin{matrix} \frac{(1-\alpha)}{\pi}\int_0^{\omega_p}\left(1-\cos\left(n+\frac{1}{2}\right)\omega\right)\left(1-\cos\left(m+\frac{1}{2}\right)\omega\right)d\omega \\ +\frac{\alpha}{\pi}\int_{\omega_s}^{\pi}\left(\cos\left(n+\frac{1}{2}\right)\omega\right)\left(\cos\left(m+\frac{1}{2}\right)\omega\right)d\omega \end{matrix} \quad , N \text{ even} \tag{3.15}$$

So we have been able to formulate the linear-phase low-pass FIR design problem in the form of an eigen problem. Given the band edges ω_p and ω_s and the parameter α, the matrix **P** can be computed. It then remains only to compute the eigenvector of a real, symmetric, positive-definite matrix corresponding to the smallest eigenvalue.

Figure 3.11 shows the response of a 16-band synthesis and analyzer prototype filter, using the eigen filter method, at a sampling rate of 16 KHz, and bandwidth 500 Hz.

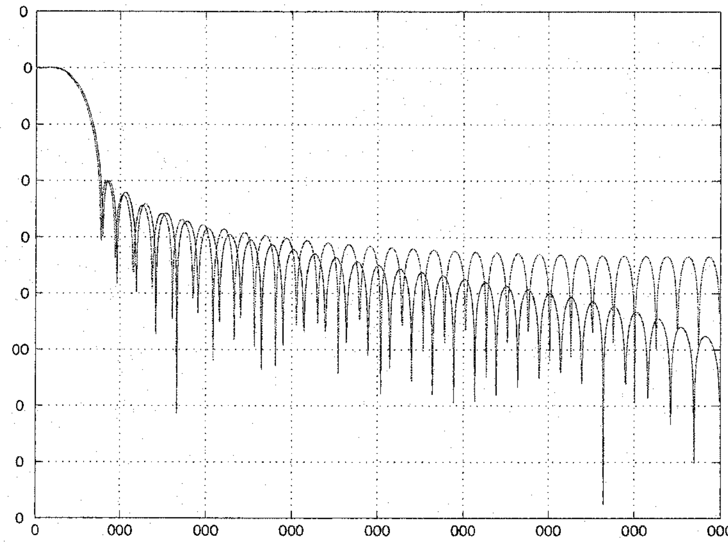


Figure 3.11 Frequency response of 63 order Eigenfilter and Kaiser Window FIR Filter

3.5 MDLNS 8 Parallel FIR Filters

3.5.1 MDLNS Filterbank Algorithm Design

As mentioned in Chapter 2, the logarithmic properties of the MDLNS allow for reduced complexity multiplication, and large dynamic range. A multiple-digit MDLNS provides a considerable reduction in hardware complexity compared to a conventional logarithmic number system (LNS) approach. In the MDLNS computation unit, as shown in Figure 2.4 on page 32, we use the MDLNS domain for the coefficient multiplication, convert to a binary representation by using a look up table and a barrel shifter, then accumulate the output using conventional binary adders. In order to design efficient MDLNS hardware, an algorithm should be designed to have all additions and subtractions operations either before or after the multiplication operations, so conversion to and from MDLNS will only have to take place twice; at the input and out. Even though the IFIR filterbank is the best algorithm for binary implementation, the requirement for feedback computation will require implementing addition in the MDLNS system, which will increase the hardware considerably. An MDLNS implementation of the DFT filterbank will need multiple conversion; consequently, neither the IFIR or DFT filterbank algorithms are suitable for MDLNS implementation.

The most suitable structure for MDLNS arithmetic is a direct parallel channel structure [20]. For this architecture, we have used MATLAB™ to design 8 parallel linear-phase FIR filters (for 8 channels), using a Kaiser-window function. In order to obtain 60 dB stopband attenuation, the order of the first to seventh filter has to be at least 73. The order of the last filter has to be at least 74. So there are 593 coefficients. The pass band ripple is ± 0.01 . There is a small deviation from perfect unity gain in the composite magnitude response, but if we set the order of all 8 filters to 74, the overall magnitude response is perfectly flat while maintaining the passband ripple at ± 0.01 . In this design the first 4 filters are symmetric with the last 4 filters; i.e., the first filter (low pass) has the same coefficient magnitudes as the last one (high pass), but with opposite signs; the second filter and seventh filters, the third and the sixth, the fourth and fifth have the same symmetry. For each filter, coefficient symmetry along the midpoint enables constant group delay, which is very important for audio signals. Figure 3.12, Figure 3.13, and Figure 3.14 show magnitude, phase, and overall response of the filterbank Using filterbank symmetry in the architecture, reduces 593 taps to 152 unique values with a reduction in the coefficient ROM size.

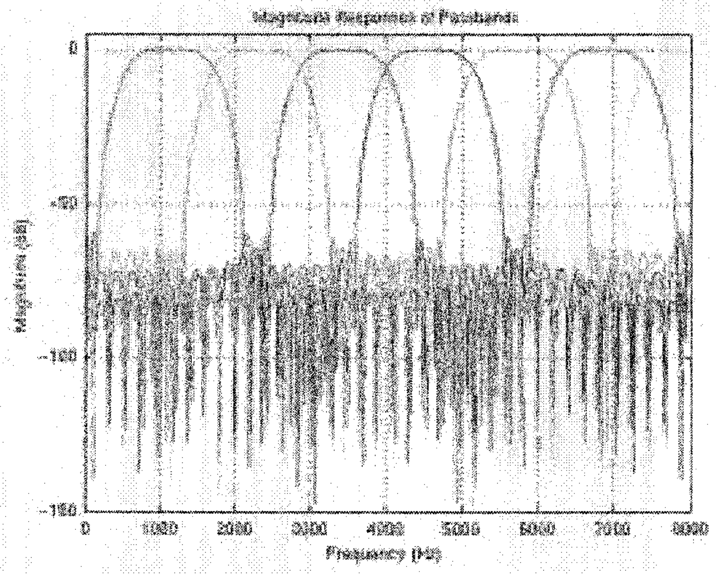


Figure 3.12 Magnitude response of 8 parallel filters

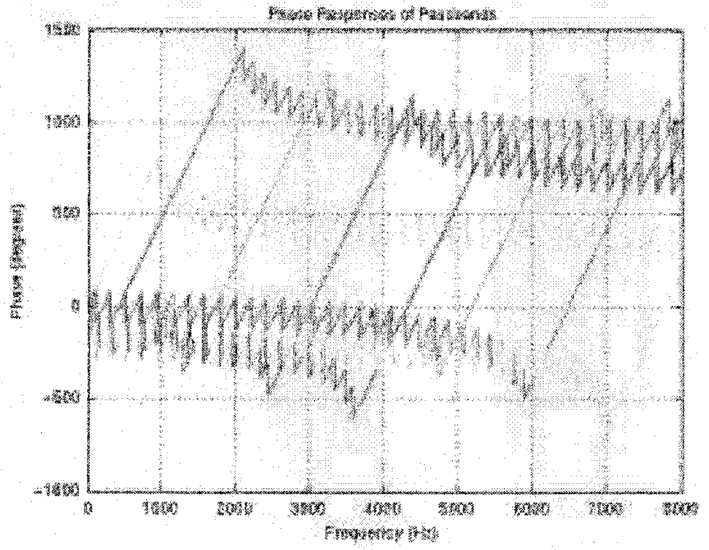


Figure 3.13 Phase response of the 8 parallel FIR filter

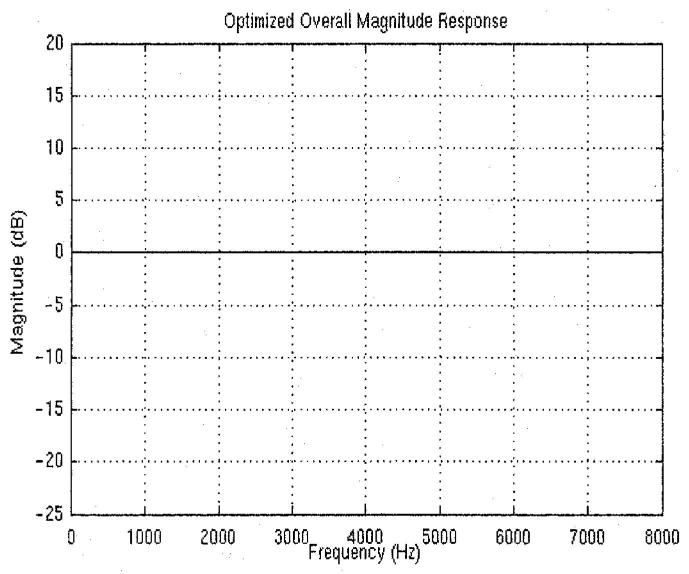


Figure 3.14 Overall magnitude response of 8 parallel filters

3.5.2 Coefficient MDLNS Mapping

The conversion, or mapping, can be performed in two different ways. The first method is to use a modified Greedy Mapping algorithm, and the second is optimal mapping using a form of linear programming (in our case an exhaustive search was conducted) [10].

The Greedy Algorithm: For a single digit, the MDLNS number closest to the target number is selected. Multi-digit DBNS works in a similar way, the first digit is selected to be as close to the target number as possible. The following digits are selected to be as close as possible to the remaining value after subtracting the values of the previous MDLNS digits.

The modified greedy mapping algorithm is very similar to the Greedy Algorithm. It was designed for use with one and two digit MDLNS mapping. For one digit it behaves exactly as the Greedy Algorithm. For two digits, it selects the two closest numbers to the target value, then fine tunes the mapping with a second digit. The best result of the two possible is then selected. This produces a more accurate mapping than the Greedy Algorithm alone.

One advantage of the modified greedy mapping method over the optimal mapping is speed. The modified greedy mapping has an $O(n)$ time complexity where n is the number of bits per exponent.

Optimal mapping has been implemented for a few sample designs using a simple exhaustive search, where the entire representation range is searched for the best possible match to the target value. This method becomes more efficient if a sorted list is used. The mapping time increases approximately as the square of representation range. We use this method to mapping our coefficients to a 2-digit 2-DLNS representation (with a ternary odd base), the binary and ternary exponents are 5-bit and 3-bit, respectively. Magnitude, phase and overall response are plotted in Figure 3.15, Figure 3.16, Figure 3.17. Stopband attenuation is 58 dB, which is similar to the 16-bit binary representation. The phase is linear, and the overall magnitude response is flat.

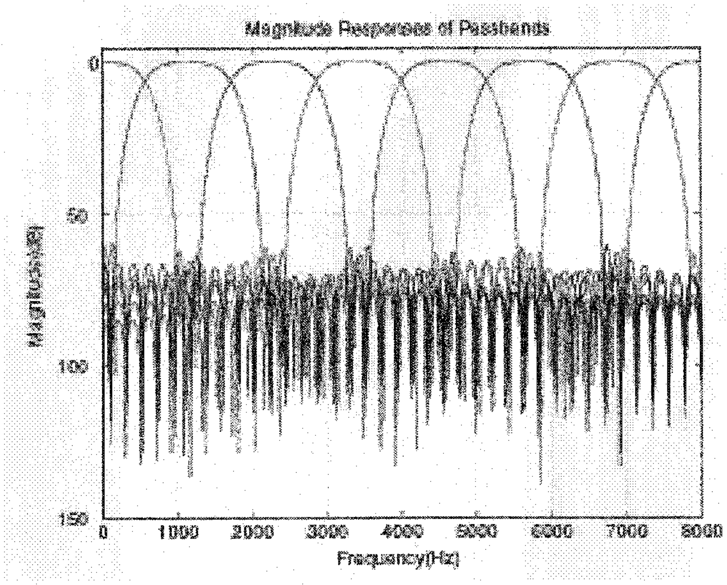


Figure 3.15 Magnitude response of MDLNS filterbank(5bit 3bit,5bit 3bit)

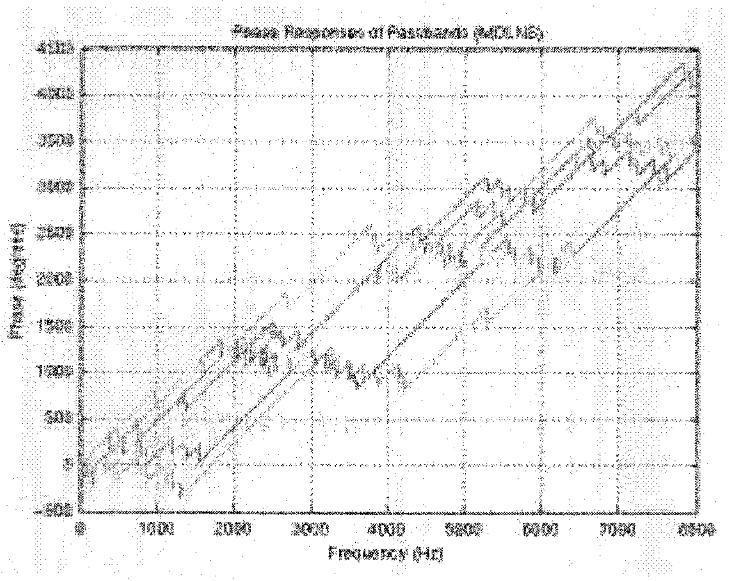


Figure 3.16 Phase response of MDLNS filterbank

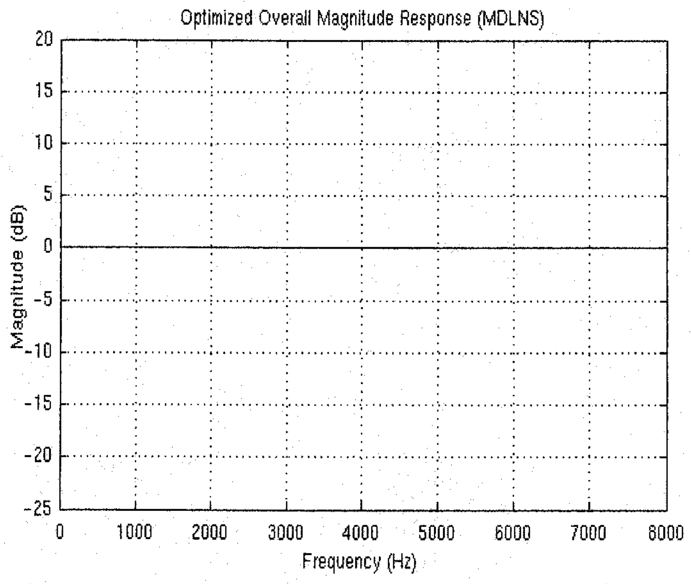


Figure 3.17 Flat overall magnitude response of MDLNS filterbank

3.5.3 Conclusions

In this chapter research on digital filterbank design techniques have been covered. Three different types of digital filterbanks have been designed and their performances are demonstrated.

The IFIR and DFT modulated filterbanks provide similar magnitude responses, but the DFT filter bank does not exhibit constant group delay; this causes distortion in the sound signal and the quality of the sound is degraded. The IFIR filterbank is certainly the best choice for a binary implementation. Using the MDLNS to implement a parallel FIR filterbank provides a similar magnitude response, linear phase, and perfectly flat overall magnitude response as the IFIR filterbank.

Chapter 4
Filterbank Processor Architectures

4.1 IFIR Filterbank Architecture

As illustrated in Figure 3.5, the IFIR filterbank consists of a tree structure of complementary interpolated linear phase FIR filters. Much effort has been devoted to reducing the number of multiplications in order to save power; many of the coefficients are zero and the nonzero coefficients in the individual IFIR filters are symmetric around the midpoint. This allows a folded implementation that effectively halves the number of multiplications [see Figure 3.2].

The modest speed requirement (sampling rate) allows for a highly sequential implementation. The algorithm can be serialized in several dimensions, using bit-serial arithmetic units and/or serialization in the time domain by mapping the arithmetic units depicted in Figure 3.2 onto a smaller set of hardware units.

The optimal choice is a dedicated processor structure (see Figure 4.1) with a single add-multiply-accumulate (AMA) data path [14], a RAM for the data samples, a ROM for the filter coefficients, and an address sequencing and control unit. Due to the folded structure of the IFIR filters, it is convenient to use a dual port RAM. Using this architecture, the processing of one input sample requires a sequence of approximately 30 AMA operations, corresponding to approximately 600 000 AMA operations per second.

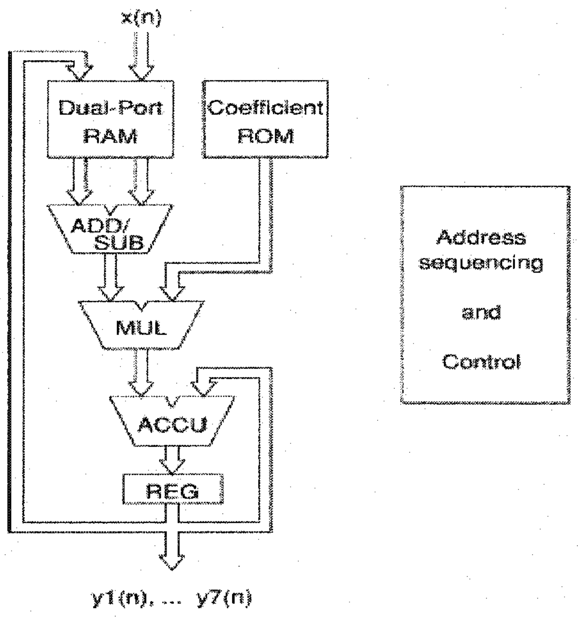


Figure 4.1 IFIR Filterbank Architecture

The main task of the address sequencing and control unit is to generate the sequence of read and write addresses for the dual-port RAM. For each IFIR filter, a portion of the dual-port RAM is administered as a cyclic buffer; when time progresses one step and a new data sample is input, it is stored in the location which holds the oldest data sample that is no longer needed. The input sample stays in this location throughout its use in computing the output stream (in order to avoid power consuming data shifts). For this reason, the addresses in a computation sequence for an IFIR filter must be offset by one from one input sample to the next.

In combination with the many coefficients being zero, this results in a very irregular address sequence. As an example, filter H7 in Figure 3.2 has 31 delay elements, and its address sequence is defined by the code fragment in Figure 4.2

```
for each input sample do
begin
  offset := offset + 1;
  write   (0+offset) mod 31;
  read    (0+offset) mod 31; read    (30+offset) mod 31;
  read  (0+6+offset) mod 31; read  (30-6+offset) mod 31;
  read (0+12+offset) mod 31; read (30-12+offset) mod 31;
  read   (15+offset) mod 31;
end;
```

Figure 4.2 The address sequence for filter H7 (indicating the address pairs for the dual-port RAM)

All IFIR filters have an odd number of delay elements, and in the actual implementation the writing of a new input sample is performed in the same step as the last read operation related to the processing of the previous input sample. Furthermore, in this last step, the data path performs a multiply-accumulate-subtract (MAS) operation, thereby producing the two outputs from the IFIR filter. This means that the data path must be able to perform both AMA and MAS operations [14].

4.2 DFT Filterbank Architecture [24]

DFT filterbank has analysis and synthesis processing as mentioned in the last chapter. Analysis processing (Figure 4.3) shifts an L-sample window along the input FIFO, R samples at a time. Following this, the signal is "folded," circularly shifted and processed with an N-point FFT. Typical values are R=8, L=128 and N=32, resulting in $N/2+1$ complex frequency domain values computed every R samples

Synthesis processing (Figure 4.4) is the inverse of analysis processing. The modified frequency domain data is processed with an inverse FFT, circularly shifted and periodically extended. Then, the synthesis window is applied and an overlap-add operation is performed. The filterbank oversampling rate (*OS*) is *OS=N/R* (i.e., FFT size / input block size). The synthesis window is also decimated by *OS*.

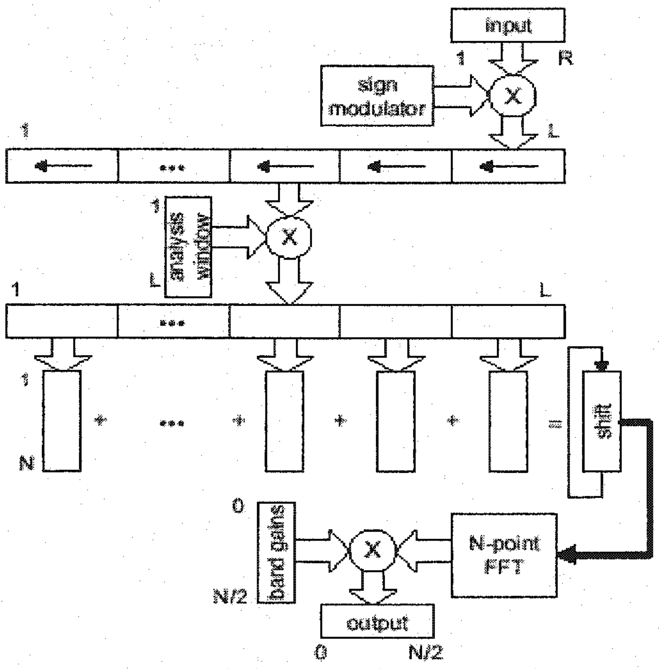


Figure 4.3 WOLF DFT analysis processing

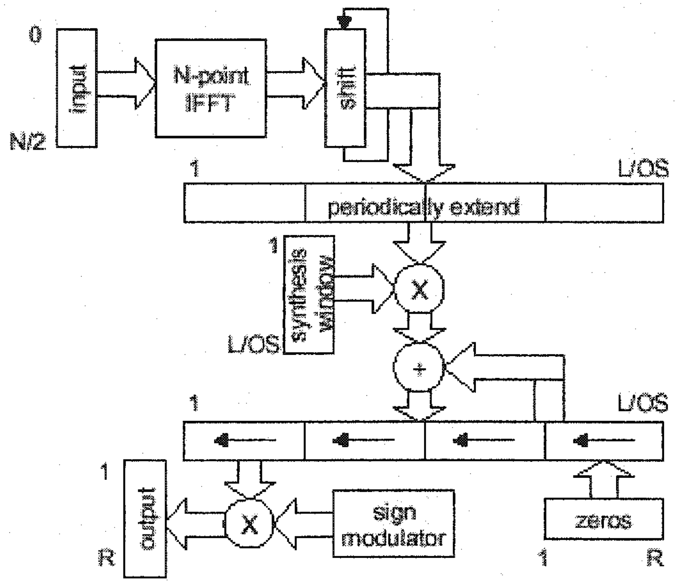


Figure 4.4 WOLA DFT filterbank synthesis processing

4.3 MDLNS Filterbank Architecture

The MDLNS filterbank is designed with 8 parallel FIR filters. The datapath architecture is shown in Figure 4.5 [20]. The number of channels needed by the MDLNS processor (Figure 2.4) is equal to the number of coefficient digits multiplied by the number of data digits. For a 2-digit system we therefore require 4-channels; though, if we use a greedy algorithm to reduce the relative magnitude of the second digit, we are able to remove the channel that corresponds to the multiplication of the two low-order digits with acceptable precision reduction [10]. This translates to three 5-bit and three 3-bit adders for a complete filter tap multiplication.

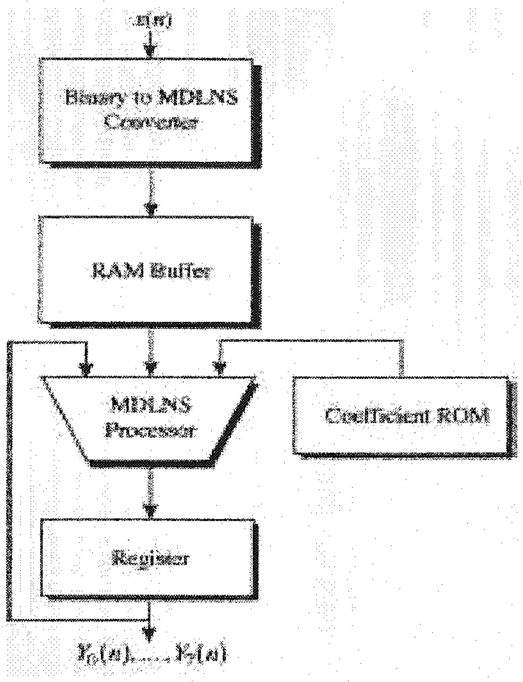


Figure 4.5 8-bank 2-DLNS filter architecture

Referring to Figure 4.5, a converter [11] outputs the data to a RAM buffer, and the ROM holds the MDLNS FIR filter coefficients. A control unit (not shown) generates the read and write addresses for the RAM and ROM, and resets the register when the computation of each filter is finished. The filterbank outputs, from $Y_0(n)$ to $Y_7(n)$ (for 8 channels), are computed and output serially for each buffered input sample. The processor requires 625 clock cycles to process each input sample, and the throughput of the system, for a 16 KHz sampling frequency, is 10MHz. For a single-digit 2-DLNS, the data and coefficient ternary exponents are 8-bits (8-8 system); For the 2-digit 2-DLNS processor we can use 5-3-5-3 coefficients. Note that the number of exponent bits are the same for each system, but the ternary ROM (shown in Figure 2.4) is reduced by a factor of $2^5=32$ and the implementation requires much smaller independent adders than the 8-bit adders of the 1-digit system.

4.3.1 Improved MDLNS Architecture

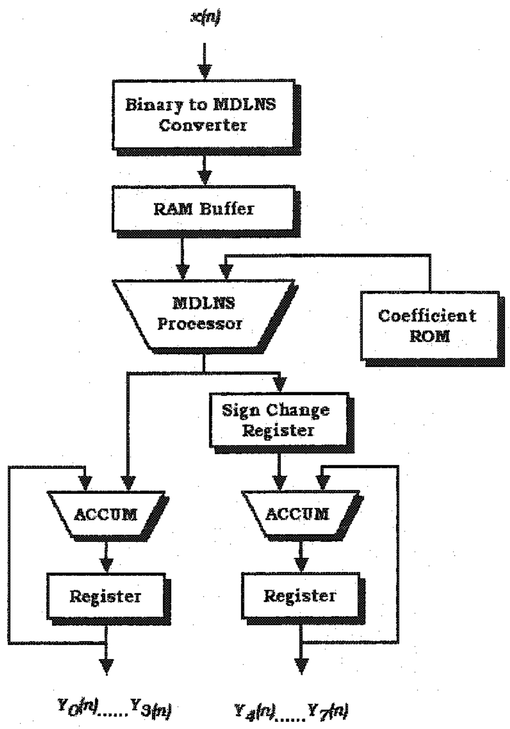


Figure 4.6 Improved MDLNS processor

If we design the each of 8 FIR filters using 75 taps, then the filterbank is made of 8 perfect symmetrical FIR filters. The filters themselves are also symmetrical. So there are only 152 coefficients instead of 593 coefficient. the coefficient ROM size is reduced. Improved architecture is shown in Figure 4.6 [21], two parallel channels are used instead of one to computer the binary output of the MDLNS processor. The first channel accumulates and outputs the first, second, third, and fourth filters serially and at the same time the second channel with a sign change register compute the fifth, sixth, seventh, and eighth symmetric filters serially. The processor requires 313 clock cycles to process each input sample. For a 16 KHz sampling frequency, the clock frequency is 5 MHz instead of 10 MHz, and so power is reduced.

4.3.2 Results and Conclusions

Three filterbank architectures (IFIR, DFT and MDLNS) are carried out. The cost of IFIR and DFT filterbanks is almost the same in terms of MPU (Multiplication per unit time) and APU (Addition per unit time) [1]. IFIR has 27 MPU and 45 APU. On the other hand the DFT filter bank has 28 MPU and 40 APU. Even though the MPU and APU numbers are close to each other. In the DFT filter bank 220 multipliers and 318 adders are needed in the hardware. Not all of them operate at the same time but their locations are independent to each other. Therefore the IFIR filter bank takes less space than the DFT filter bank [1]. The IFIR and MDLNS filterbank architectures looks similar, but actually the MDLNS filterbank removes multiplication, which reduces the power consumption unit compared to a binary implementation. The read and write addresses of the RAM used in the IFIR filterbank are quite irregular due to the fact that many of the coefficients are zeros. This requires a more complex control unit for the IFIR filterbank compared to that required for the MDLNS filterbank. Comparing 1-digit and 2-digit implementations, the full 2-digit MDLNS filterbank produces a much smaller ternary exponent. This provides a large reduction in the area consumed by the look up table ROM.

Chapter 5

Physical Implementation of Filterbank Processor

5.1 Introduction

ASICs (application specific integrated circuits) is one of the fastest growing segments within the semiconductor market. A system realization incorporating ASICs offers many advantages, and an ASIC can replace a large number of discrete TTL or MOS logic components and dramatically lower the total system cost due to the minimization of chip size and power requirements. ASICs also make possible a vast variety of applications with functions and complexities that cannot be realized with traditional implementation methods, such as a CIC hearing aid. ASIC-based systems are an entirely new approach to furnish the IC with specific functionality while also aiming at faster turnarounds.

The multidimensional logarithmic number system (MDLNS) filterbank processor has been shown to be theoretically sound. This was shown through theory, mathematics and MatLab simulations. To fully test and prove the theory behind the MDLNS filterbank, a physical microchip was built and tested. The designed filterbank consists of eight 75 tap MDLNS filters with full 2-digit representation (i.e., 2 digits for both the data and filter coefficients), We used 3 as the second base in our initial design, with a range for the binary and ternary coefficient exponents 5 bits and 3 bits, respectively. We have discovered, however, that by optimizing the second base, it is possible to find a 6-2 6-2 representation which is more accurate. Using this new representation, we can save one bit in the second base exponent; in this case the 2nd base has the value 1.283081 [21]. 6 bit binary and 5 bit secondary exponents are used for the each of the data digits. The design methodology is shown in Figure 5.1, as prepared by the Canadian Microelectronics Corporation (CMC). The Hardware Description Language (HDL) used for the design simulation and synthesis was Verilog. The work presented here was the combined effort of the author and Mr. Roberto Muscedere, who is using the design to evaluate novel MDLNS converter architectures [26].

5.2 Design Flow

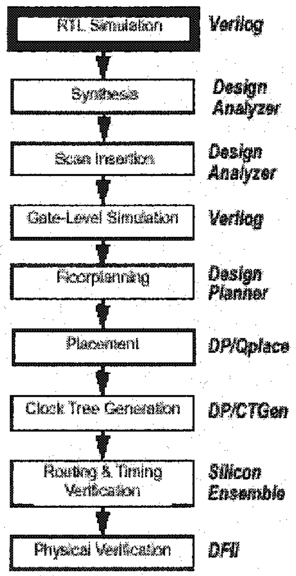


Figure 5.1 Digital ASIC Design Method

Cadence and Synopsys are tools used for the design flow. A CMC design kit for Cadence and Synopsys tools typically contains technology files and device libraries which, when combined with the CAD tool, enable design of an integrated circuit in a particular process technology. The TSMC 0.18-micron CMOS technology has been used for this design.

5.2.1 RTL Simulation

RTL (register transfer level) simulation, with the appropriate testbench, enables the functionality of a design to be checked against the specifications. Verilog and VHDL are the two languages of choice. Verilog is similar to C, and is the best choice for simulation, because the simulation results are easier to interpret and understand than simulation using VHDL. Another advantage of verilog is that the testbench can be re-used later on.

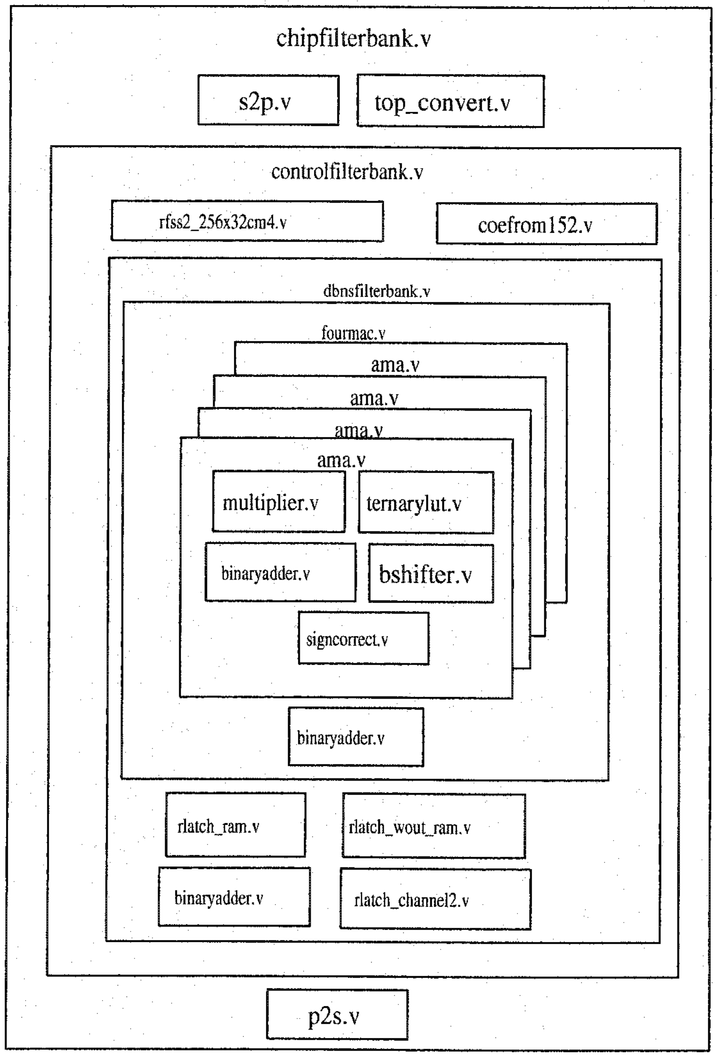


Figure 5.2 Hierarchal diagram for HDL description of a MDNS filterbank

Users may also use VHDL at this point in the flow, but two testbenches will have to be created, one in VHDL for RTL and gate-level simulation, and one in Verilog for pre-layout and post-layout full-timing simulation. The Verilog code used in our chip design was built using a hierarchical structure. Using a top-down approach, the overall filterbank was described in chipfilterbank.v. Next four subblock **s2p.v**, **top_convert.v**, **controlfilterbank.v** and **p2s.v** were described, and finally the individual components of each block were described. Figure 5.2 shows the Verilog hierarchy.

s2p.v (serial to parallel input block): The input serial port obtains the input binary data from the ADC serially and is a serial in parallel out (SIPO) register. Input data from the port "inputbit" is latched into the register at the positive edge of the clock and enable signal. 16 bits of data are latched and sent to the binary to MDLNS converter block. 16-bits of input data are processed every 313 clock cycles.

p2s.v (parallel to serial output module): It is basically a Parallel in Serial Out Shift (PISO) Register that is used to shift the data out in a serial fashion. Once the computation of two sets of filter data from two parallel channels is completed, output data is shifted one bit by one bit to the output ports "sout1", "sout2". Both s2p.v and p2s.v are used to reduce the I/O pin numbers, so that the entire design is core bound rather than I/O bound (where the size and number of connection pads dictate the size of the layout).

top_convert.v (Binary to MDLNS converter module): The converter receives an activate signal from the control unit and begins to convert the 16 bit binary input data to 2-digit 2-DLNS. For each digit, the sign is two bits (00 represent 0, 01 represents positive, 11 represents negative), the binary exponent uses 6-bits, and the second base exponent uses 5-bits. In total, 26 bits of MDLNS data are send to the RAM at the positive edge of the clock. 15 clock cycles are needed for completion of one input data conversion.

controlfilter_ram.v (MDLNS filterbank module): This includes a RAM for holding input MDLNS data, a ROM for MDLNS coefficients, and the filterbank data path and control signals.

Control signals: This is a base counter fed by the clock. The count is from 0 to 312 which is the number of clock cycles needed to process one sample of input data. Other control signals, such as the *activate* signal for the converter, *clear* signal for the accumulation register, *ready* signal for the output and other counters that sequentially generate the addresses for the ROM and RAM are driven by this base counter. Major subblocks of this module are explained below:

coefrom152.v: is a ROM that holds 152 MDLNS coefficients. Each coefficient is represented by a 2-digit MDLNS: 2 sign bits, 6-bits for the binary and 2-bits for the second base exponents for each of the coefficients. Totally 20 bits are used to represent one coefficient, 38 unique coefficients are stored for each 75 tap FIR filter, and only the first four filter coefficients are needed to be stored due to the symmetry of the filter design. The size of the ROM is 152 x 20, and a counter generates its read address. The address increments to 38 then decrements to 0 to finish the computation of the first filter, it is then preset to 39 for the first coefficient of the next filter... and so on.

rfss2_256x32cm4.v is a standard RAM cell (256 x 32) from the "black box" library provided by Virage Logic Corporation. The implementation of a 75 tap FIR filter requires windowing of 75 data samples, so that the size of the RAM required is 75 x 26. Unfortunately we were unable to find a RAM design of this size in the library and so we had to use a larger design. The same 75 data samples are used over all eight filters, so only one RAM is required.

The RAM read and write addresses and the input data are established at the positive edge of the clock, and the read and write of the MDLNS data is performed at the negative edge of the clock to ensure that all the RAM input signals are stable before they are latched. A counter sequentially generates the read and write address for the RAM. New input data are written to the RAM location which holds the oldest data.

dbnsfilterchip_ram.v: datapath of the filterbank. It consists of 4 MDLNS processor channels for a full 2-digit computation. The following are contained in each MDLNS processor: (ama.v), mutiplier.v (two adders); a 32 x 17 ROM (ternarylut.v); a shifter (bshifter.v) and adders (binaryadder.v); a sign correction module (signcorrect.v). The output of this stage is a binary number. There are two channels that accumulate and output two symmetric filters at the same time.

In the testbench, an input chirp signal is provided from 0 to 8KHz, over 18,000 samples. This signal is supplied to the simulation in a 16-bit binary format, where each sample is sent to the chip in a bit-serial mode. The final 8 outputs of the filterbank Verilog simulation are plotted in Figure 5.3. The frequency range is the same as the input signal from 0 to 8KHz. The magnitude response is normalized to one

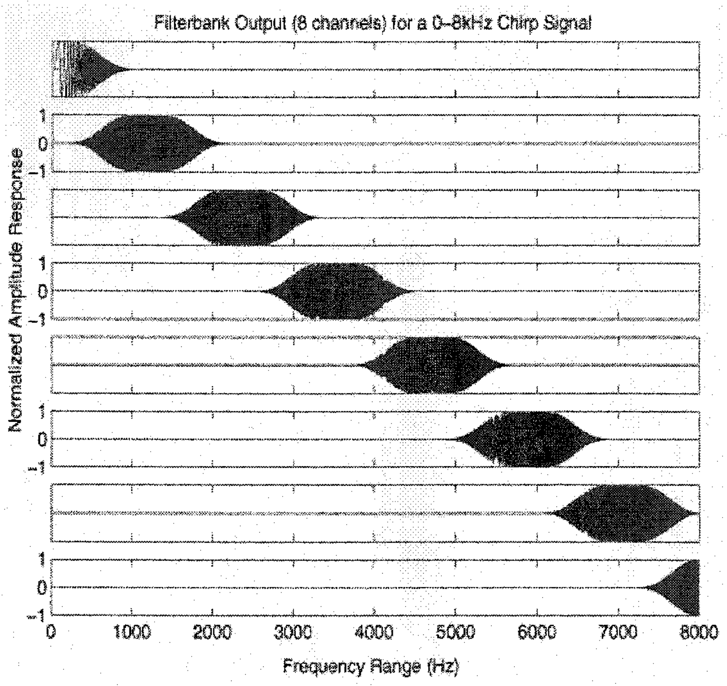


Figure 5.3 Verilog simulation of the entire MDLNS filterbank

5.2.2 Logic Synthesis

Design synthesis is what allows for rapid chip design. The synthesis tool Synopsys Design Compiler takes the RTL verilog code file, constrains the file and compiles it into a technology-specific gate-level implementation. This allows the designer to describe a system using a programming like interface and yet produce a hardware description. Only RTL level code needs to be described, such as addition, multiplication or shifting. Synopsys handles the conversion of this description into gates or other cells supplied by the library of standard cells. Notice that Verilog RTL code for simulation may not suitable for synthesis, so we must modify the simulation RTL code to synthesizable RTL code.

TIn our chosen technology we will invariably meet the time constrain for this modest sampling rate design, so we use an incremental compiler with a power constraint of 600 Wto minimize the power. The output format from Synopsys is a Verilog gate level file for use with the Cadence tool Design Floorplanner, and a system timing file (.sdf) used in Silicon Ensemble for routing.

There was one major problem encountered during synthesis. There were no ROM libraries for use with Synopsys. This meant that the ROMs in the design, one for the coefficients and one for each MDLNS processor cell, had to be created from combinational logic. This greatly increased the area of the components that contain ROMs.

Four gate level blocks are shown in the following figures. Library cells are yellow, buses are blue.

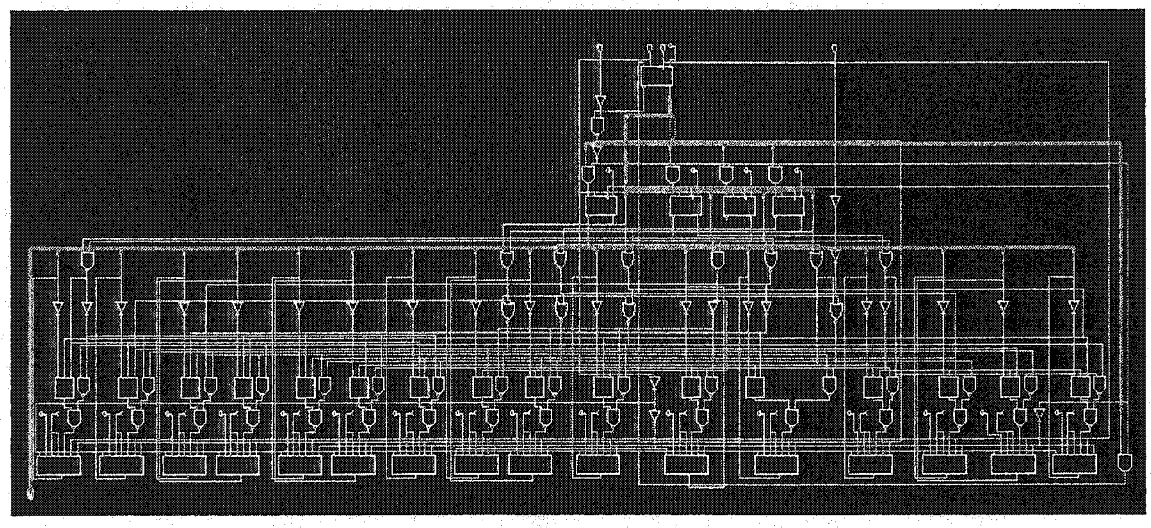


Figure 5.4 Gate level input data serial to parallel converter block (128 cells)

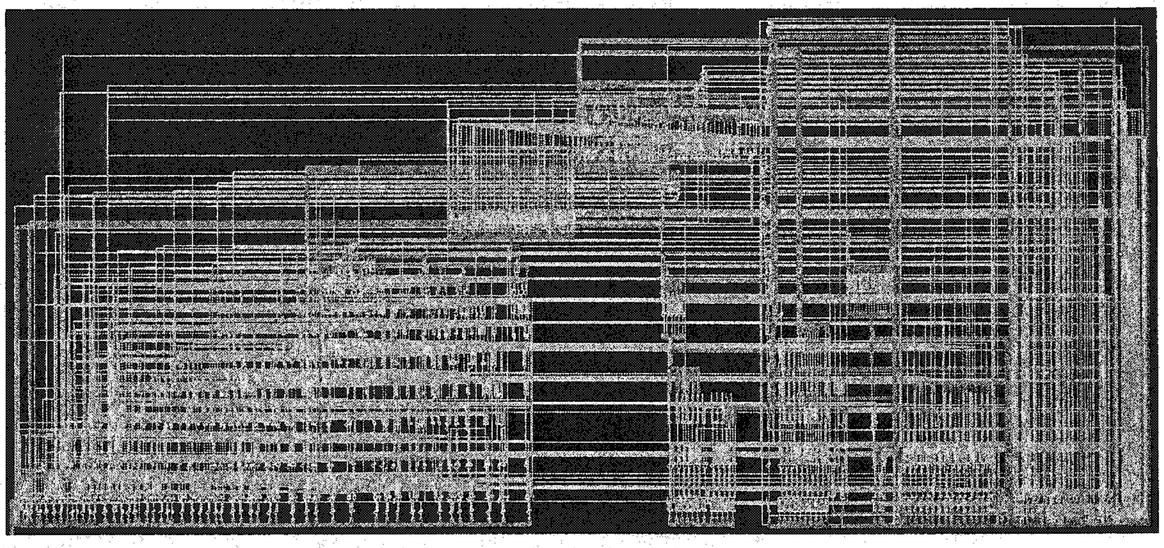


Figure 5.5 Gate level binary to MDLNS converter block (762 cells)

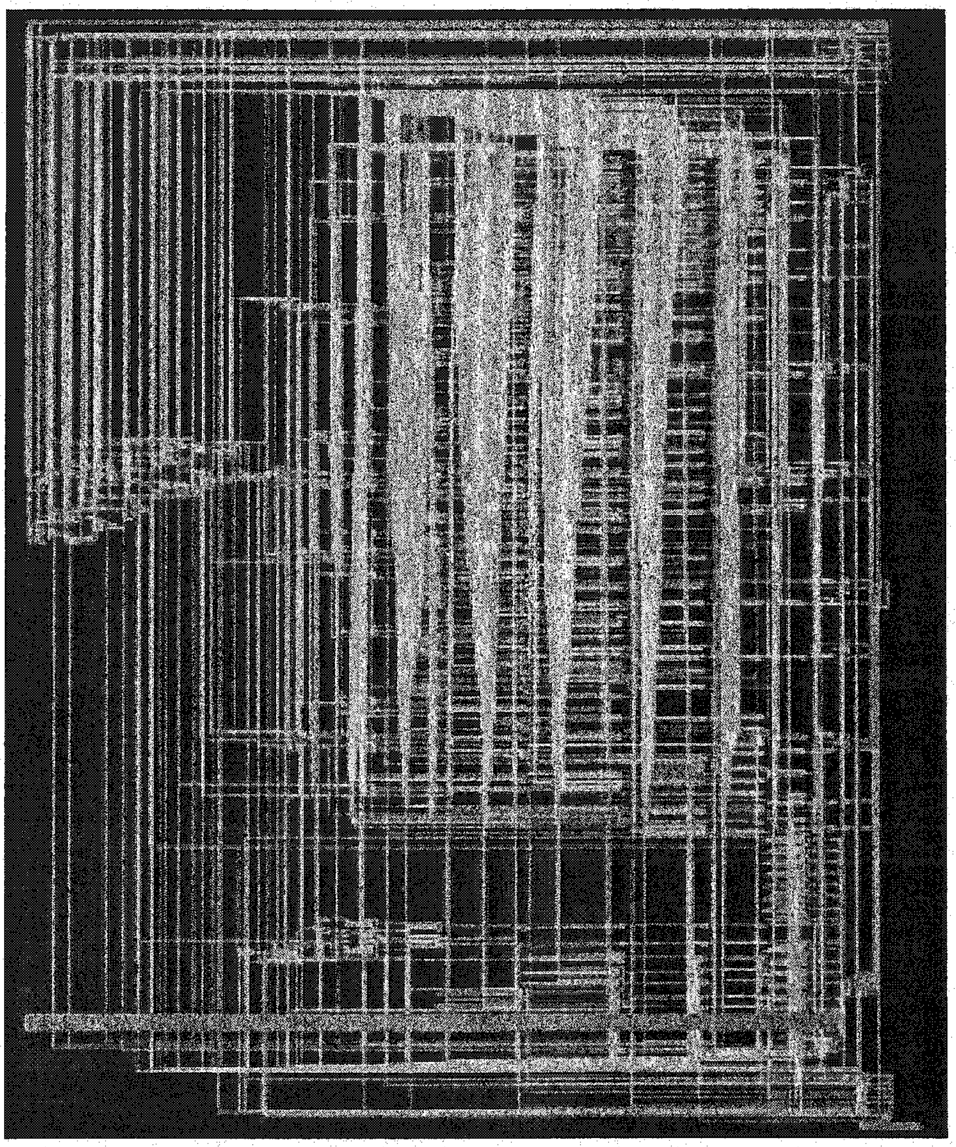


Figure 5.6 Gate level MDLNS filterbank block (1883 cells)

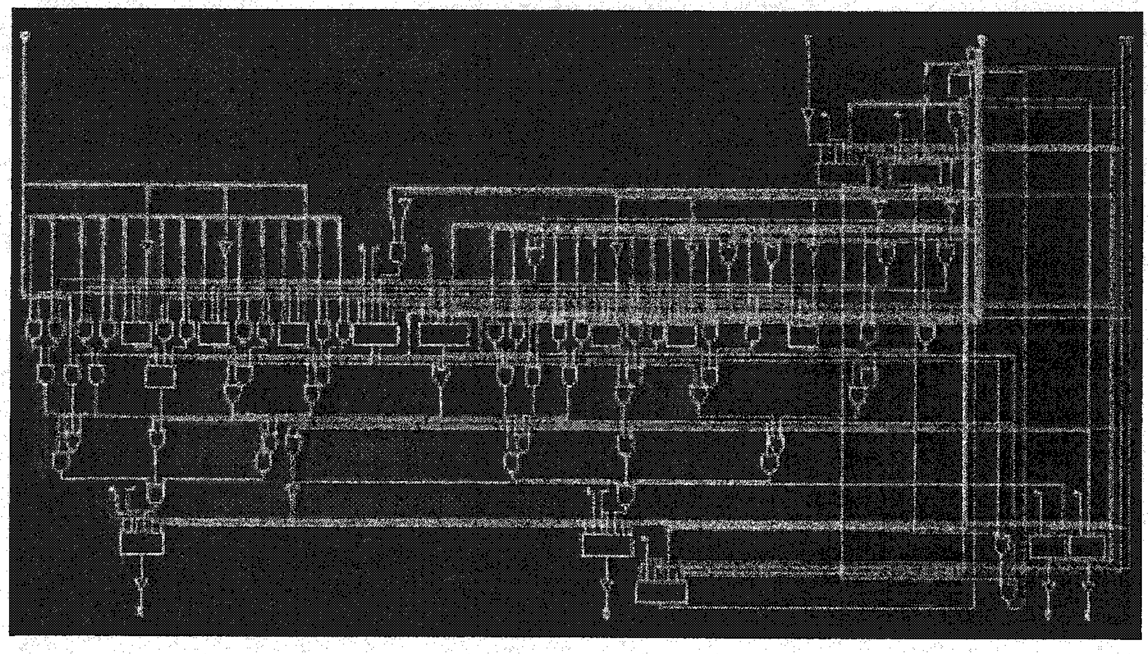


Figure 5.7 Gate level output data parallel to serial converter block (69 cells)

5.2.3 Gate-Level Simulation

A gate-level simulation was used to check the functionality of the structural netlist against the RTL simulation. The testbench used previously for the RTL simulation is used here.

5.2.4 Floorplanning and Timing-Driven Placement

The physical portion of the IC design starts from here. The Cadence tool Physical Design Planner (PDP) was used for these steps. Floorplanning is used to create a floorplan for the design including: a default group of cells; I/O ring connected by abutment; defined placement site for all cells. RAM cells are manually placed at this step.

Placement is used to place the core cell in order to optimize the connectivity between groups and blocks. The automatic placement tests potential placements for the design and tries to optimize the placement for overlap removal, routing congestion balancing, power balancing, wire length, timing assurance, and wired-logic distance constraints. An SDF constraint file is used to raise the criticality of nets that are part of the delay paths.

The Figure 5.8 show the design after placement. A total of 32 pads are evenly spaced around the square chip (8 pads per side): 8 I/O pads; 4 pair of core power pads which provide power to the core of the chip; 8 pair of ring power pads to provide power to the other I/O cells, and drive signals off-chip. 5 feeders are inserted between pads to make sure core area have enough space. Red line are rows for placing the cell. Black rectangle at up left of core is the RAM. Yellow boxes are placed library cells.

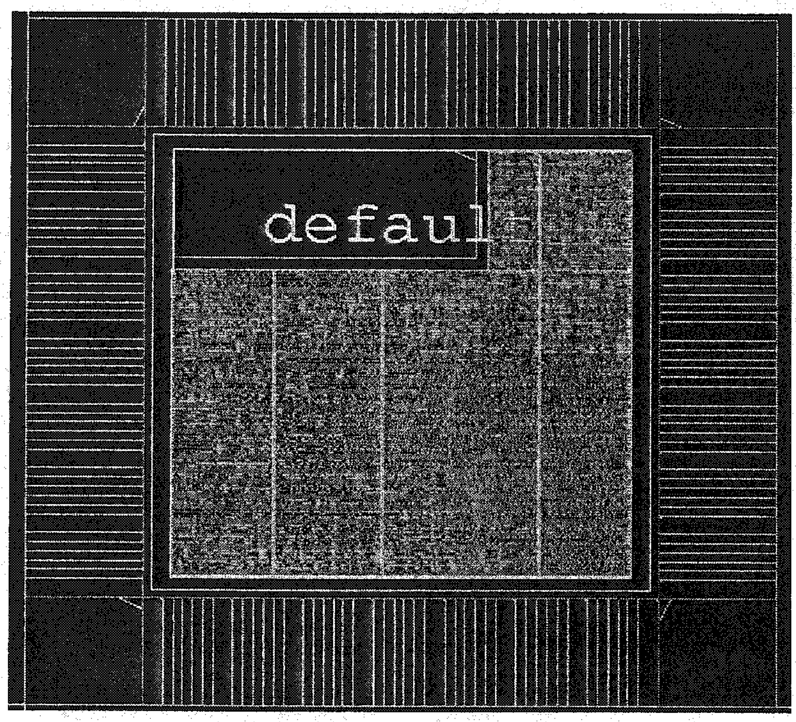


Figure 5.8 Design of MDLNS filterbank processor after placement

5.2.5 Clock Tree Generation

A clock tree has to be built when a large number of cells are clocked by a single driver cell in order to minimize, or accurately control, the signal skew at the clocked cell's inputs. CTGen is the tool for this step. A physical library, which includes timing data in a Timing Library Format (TLF), is also used here. CTGen adds clock buffer cells and nets to create a balanced clock tree. All modifications to the netlist are saved in a DEF file for back-annotation to the original netlist through Engineering Change Orders (ECOs). An analysis of the timing reports, produced by CTGen, is carried out before proceeding to the next step. CTGen is used to create a golden Verilog netlist for Layout versus Schematic (LVS) verification.

The design, after the clock tree generation, is shown in Figure 5.9. The white cells in the core area are clock buffer cells added by CTGen.

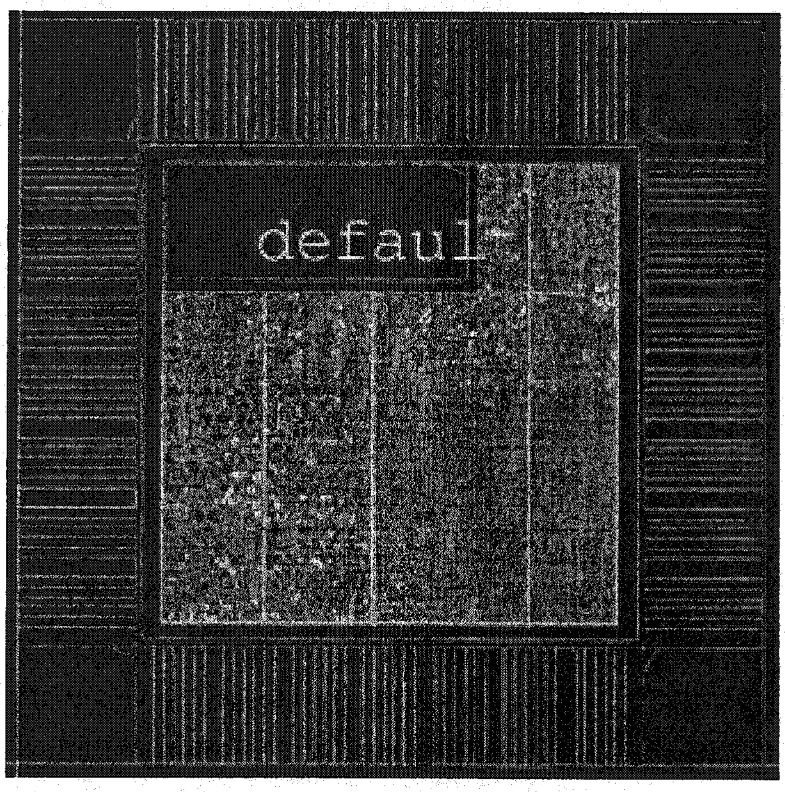


Figure 5.9 Design of MDLNS filterbank processor after CTGen

5.2.6 Pre-Route Golden Verilog Netlist Verification

This step is needed to ensure that the reference Verilog netlist, used by Diva for LVS purposes, is functionally correct. A full-timing Verilog simulation was performed using estimated or actual RCs to verify gate-level timing. Then this Golden Verilog netlist was imported into Silicon Ensemble environment for Routing.

5.2.7 Routing

This step includes global and final routing using the tool Wrout. Global routing usually consists of a coarse regular wiring layout based on obstructions resulting from special wiring, clock wiring, and placement. Final routing creates the detailed regular wiring layout as shown in Figure 5.10.

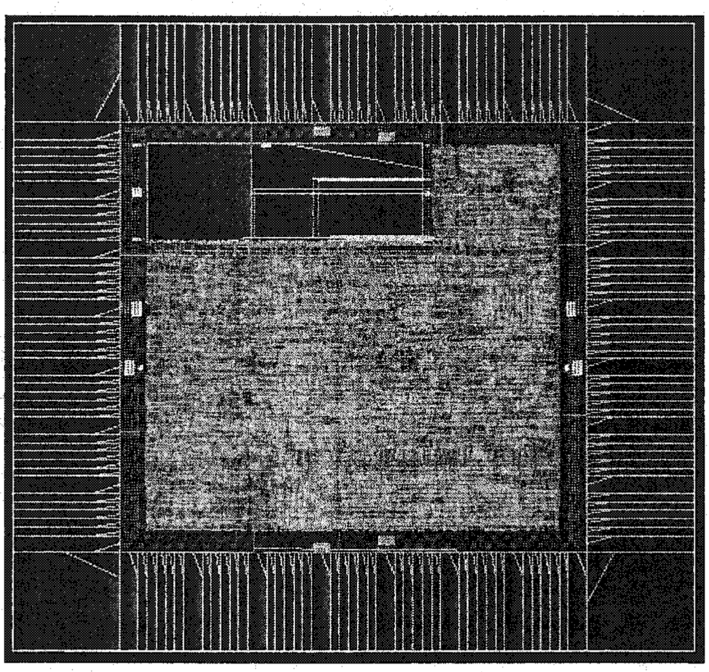


Figure 5.10 Design of MDLNS filterbank processor after routing

5.2.8 Post-Layout Static Timing Analysis

Here we use Silicon Ensemble to create the SDF files with accurate timing information needed to perform post-layout simulation and timing verification by back-annotating the SDF file. Pearl is used to perform the static timing analysis.

5.2.9 Physical Verification (DRC & LVS)

It is very important to run DRC and LVS on the layout to be sure that the connectivity, the geometry and the spacing are correct, and the layout matches the schematic. This step includes a flat extraction of the layout.

5.3 MDLNS Filterbank Processor Chip

The design layout of the fabricated chip, using TSMC 018 micron CMOS technology, is shown in Figure 5.12. Figure 5.11 is the block graph of the MDLNS filterbank chip, The blocks are a) MDLNS filterbank (Blue, 1883 cells); b) Binary to MDLNS converter (yellow,762 cells); c) Input block: serial to parallel (red, 128 cells); d) Output block: parallel to serial (green, 69 cells), and a buffer RAM (black rectangle)

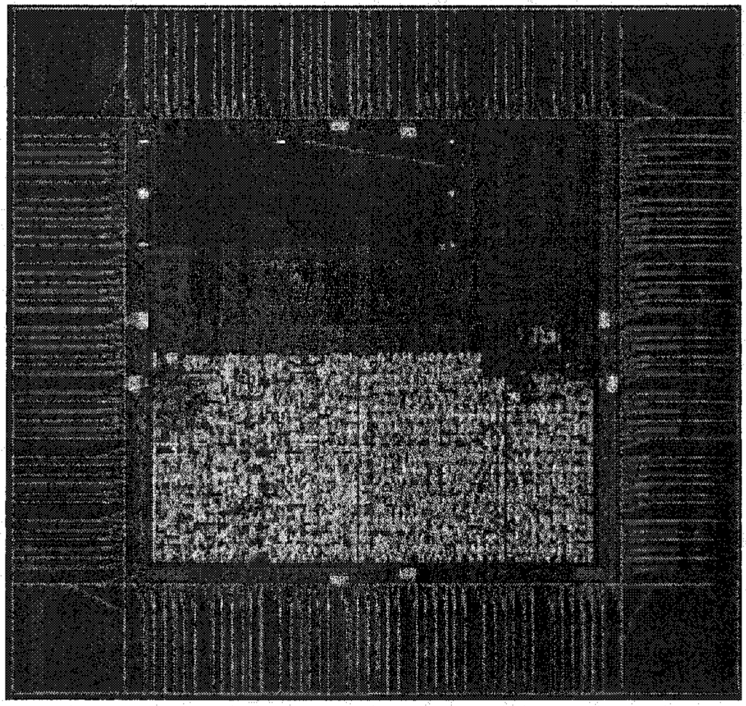


Figure 5.11 Block graph of MDLNS filterbank chip

The die size is 1490 X 1490 m^2, and core area is about 920 x 920 m^2. The entire MDLNS filterbank (blue in Figure 5.11) is about 920 x 500 m^2 including the RAM (141,047 m^2).

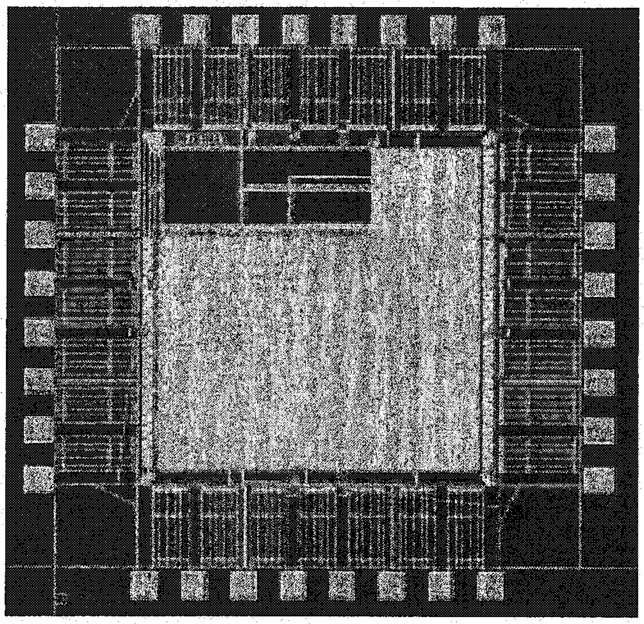


Figure 5.12 Layout of the MDLNS filterbank chip

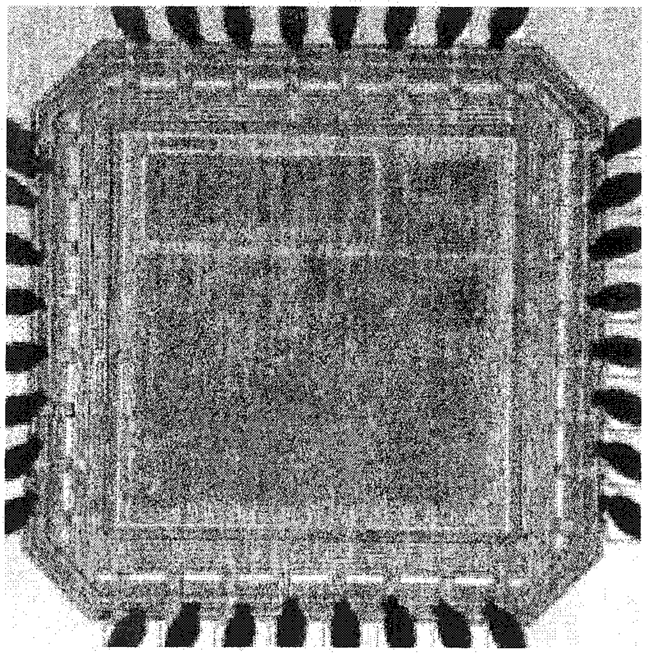


Figure 5.13 Micrograph of the fabricated MDLNS filterbank processor chip

Chapter 6

MDLNS Filterbank Processor Analysis and Comparison

6.1 Introduction

The previous chapters have presented three forms of filterbank algorithms, architectures and a comparison. This chapter covers quantization noise comparison between the MDLNS filterbank and a binary implementation, and an area and power comparison between these three forms of filterbank.

6.2 Quantization Noise Comparison

For a direct comparison of the MDLNS filterbank processor with a more conventional implementation, we use the binary number system to implement an 8 parallel FIR filterbank algorithm with a similar dedicated processor structure (a single add-multiply-accumulate (AMA) data path).

For the MDLNS filterbank, we use two digits (5-4, 5-4 represent the input data, and a (5-3, 5-3) representation for the coefficients. The binary representation uses 16-bits.

Figure 6.1 to Figure 6.8 shows the quantization noise of 8 channels of the binary implementation.

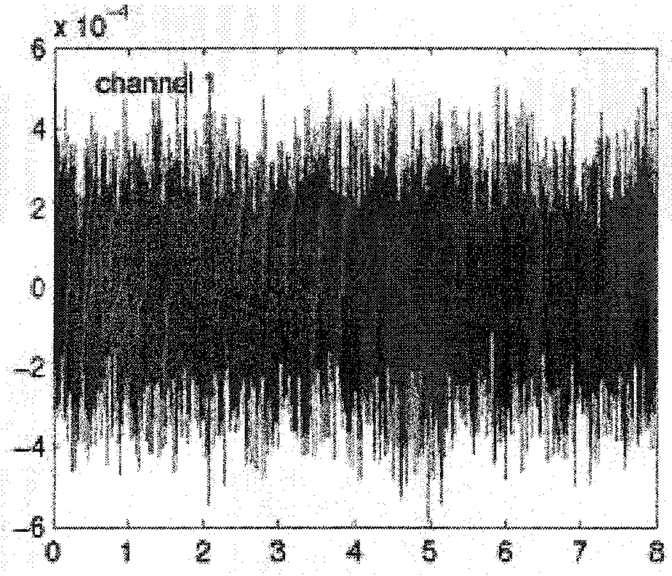


Figure 6.1 Quantization noise of 16 bit binary system: channel one

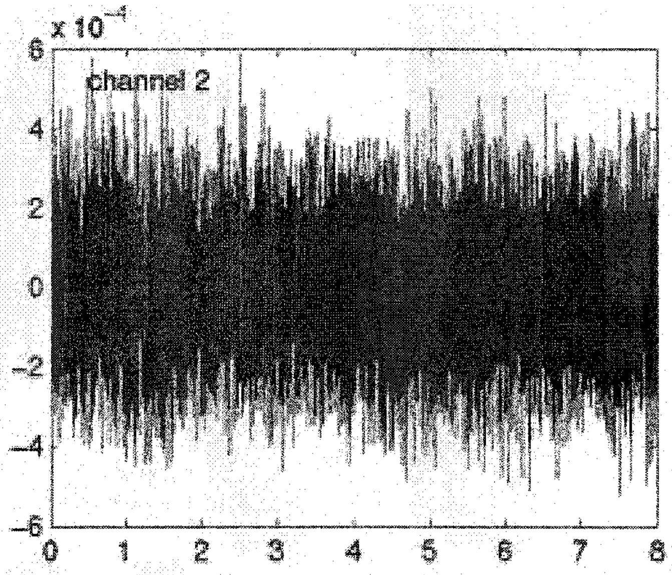


Figure 6.2 Quantization noise of 16 bit binary system: channel 2

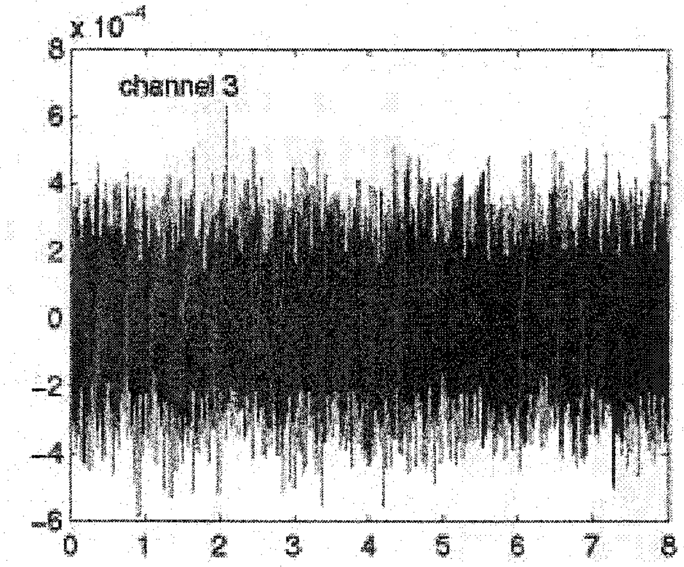


Figure 6.3 Quantization noise of 16 bit binary system: channel 3

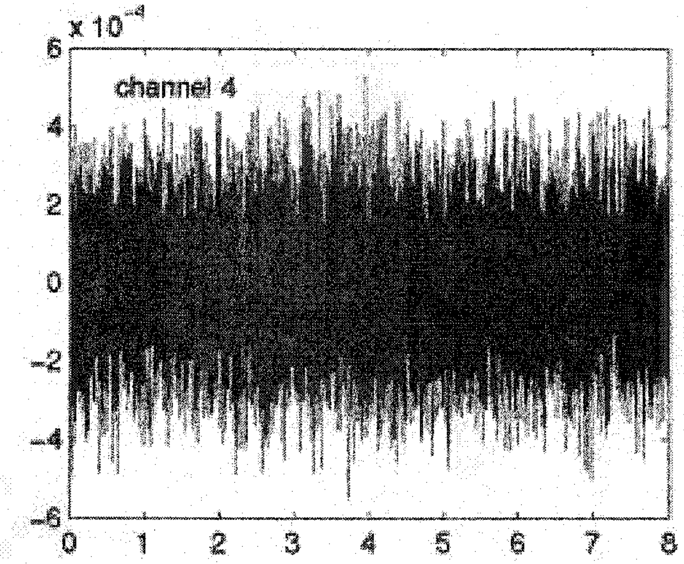


Figure 6.4 Quantization noise of 16 bit binary system: channel 4

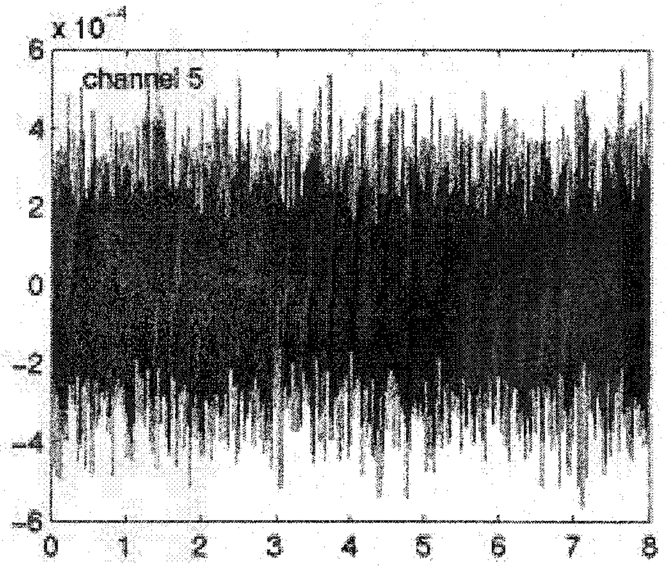


Figure 6.5 Quantization noise of 16 bit binary system: channel 5

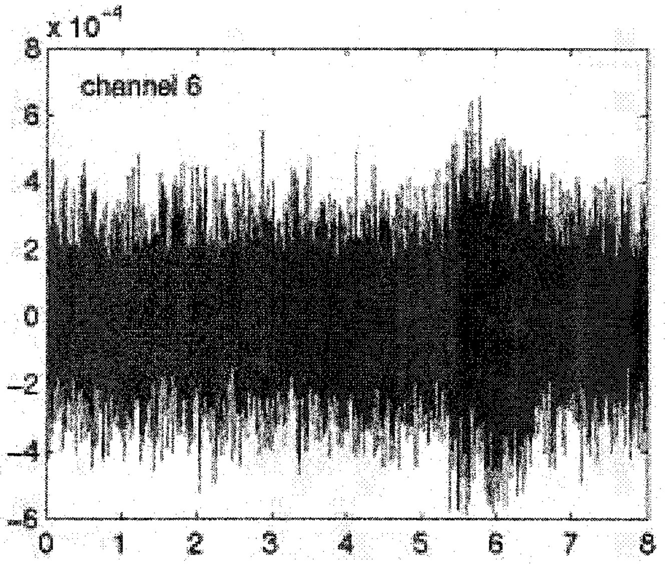


Figure 6.6 Quantization noise of 16 bit binary system: channel 6

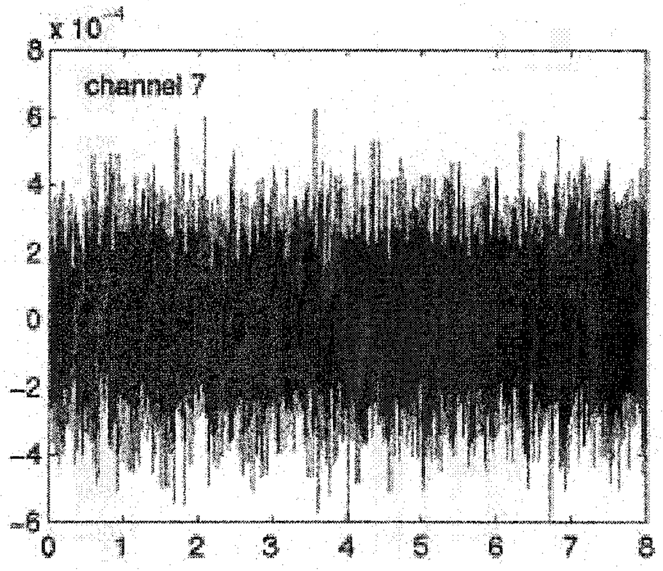


Figure 6.7 Quantization noise of 16 bit binary system: channel 7

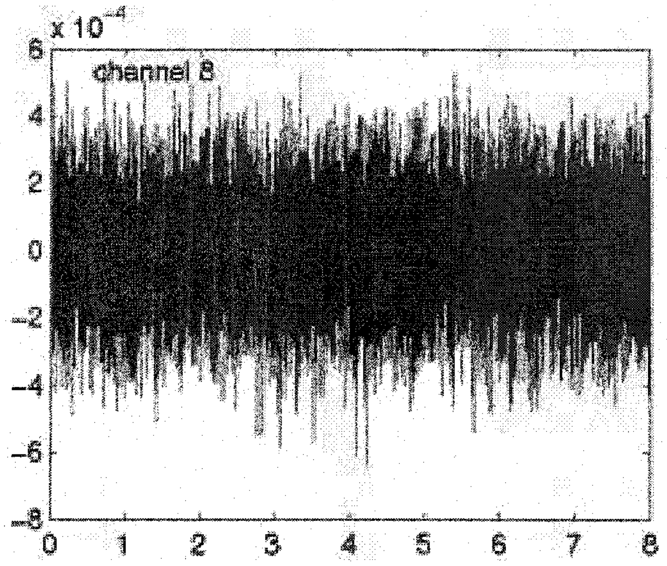


Figure 6.8 Quantization noise of 16 bit binary system: channel 8

Quantization noise range of 16 bit binary is $\pm 8\times 10^{-4}$.

The following figures show MDLNS quantization noise, The Coefficient representation is (5 3 5 3), and for the first 8 figures the input data are (5 4 5 4), then for the remaining 8 figures, the input data are (5 5 5 5).

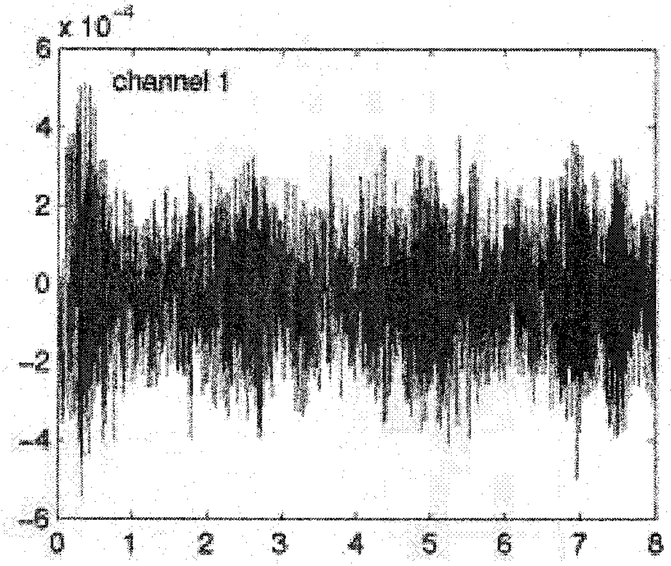


Figure 6.9 Quantization noise of MDLNS filterbank channel 1(coefficient 5 3 5 3, input data 5 4 5 4)

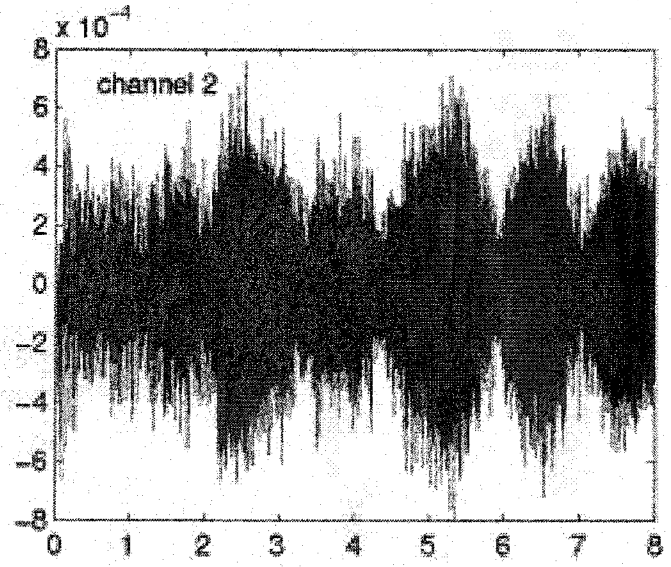


Figure 6.10 Quantization noise of MDLNS filterbank channel 2 (coefficient 5 3 5 3, input data 5 4 5 4)

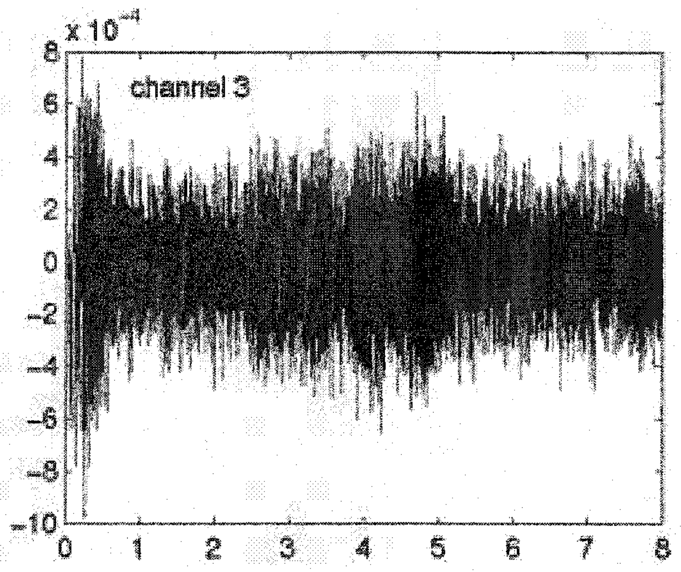


Figure 6.11 Quantization noise of MDLNS filterbank channel 3 (coefficient 5 3 5 3, input data 5 4 5 4)

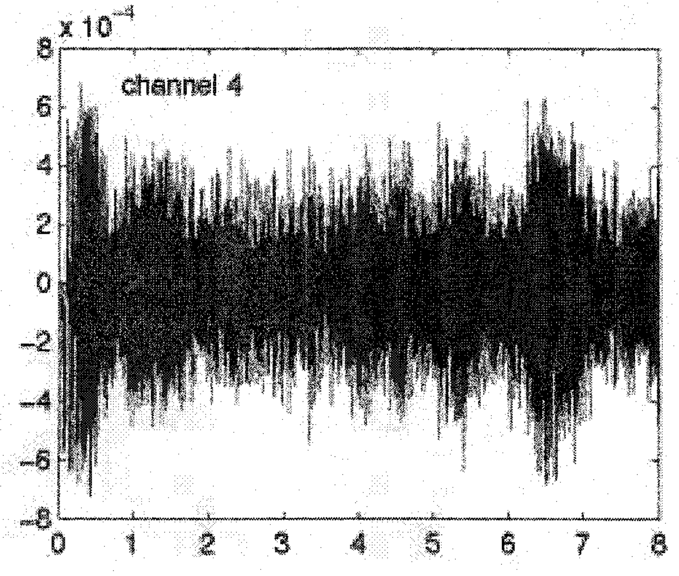


Figure 6.12 Quantization noise of MDLNS filterbank channel 4 (coefficient 5 3 5 3, input data 5 4 5 4)

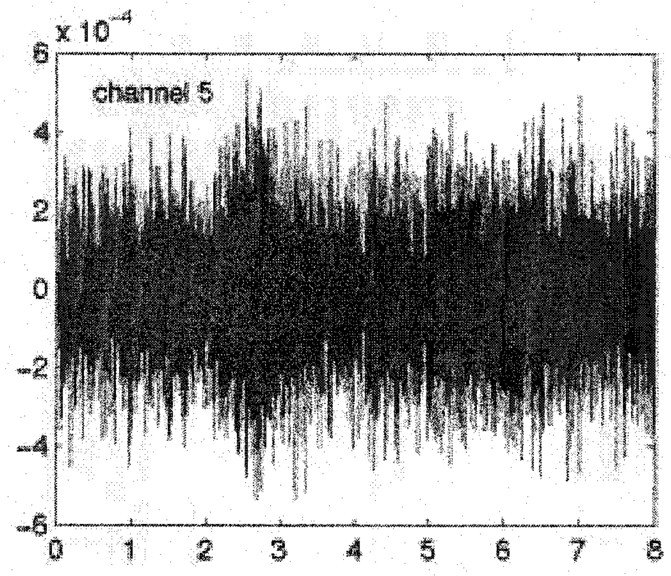


Figure 6.13 Quantization noise of MDLNS filterbank channel 5 (coefficient 5 3 5 3, input data 5 4 5 4)

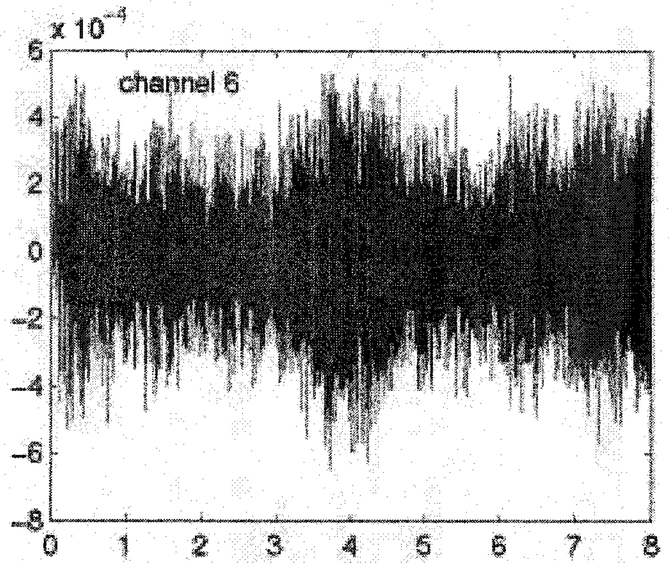


Figure 6.14 Quantization noise of MDLNS filterbank channel 6 (coefficient 5 3 5 3, input data 5 4 5 4)

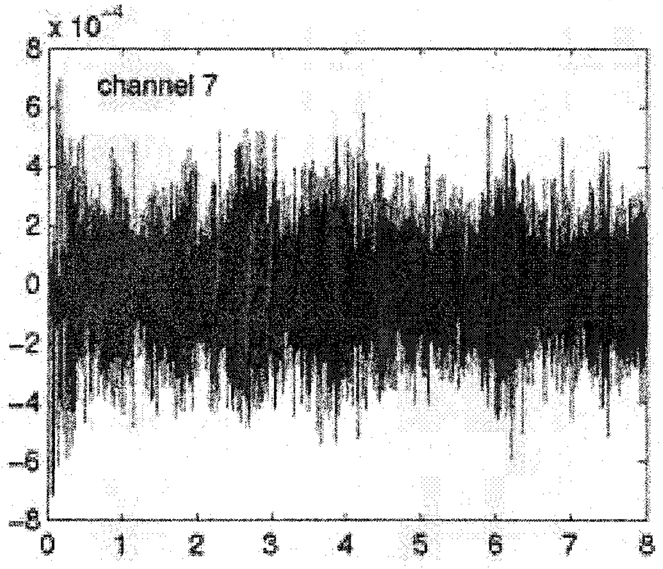


Figure 6.15 Quantization noise of MDLNS filterbank channel 7 (coefficient 5 3 5 3, input data 5 4 5 4)

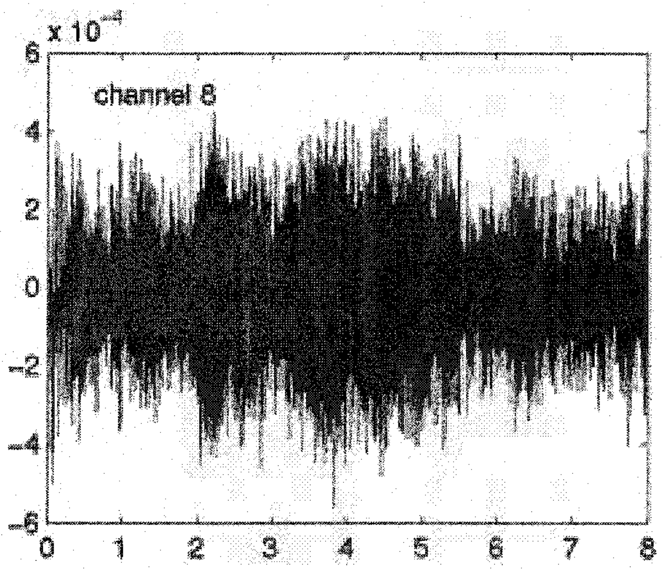


Figure 6.16 Quantization noise of MDLNS filterbank channel 8 (coefficient 5 3 5 3, input data 5 4 5 4)

The noise range of the (5 4 5 4) input data is $\pm 8 \times 10^{-4}$.

.

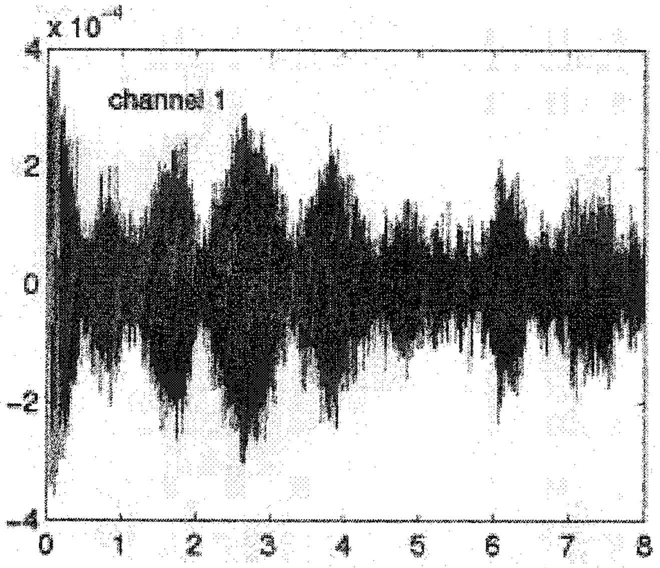


Figure 6.17 Quantization noise of MDLNS filterbank channel 1 (coefficient 5 3 5 3, input data 5 5 5 5)

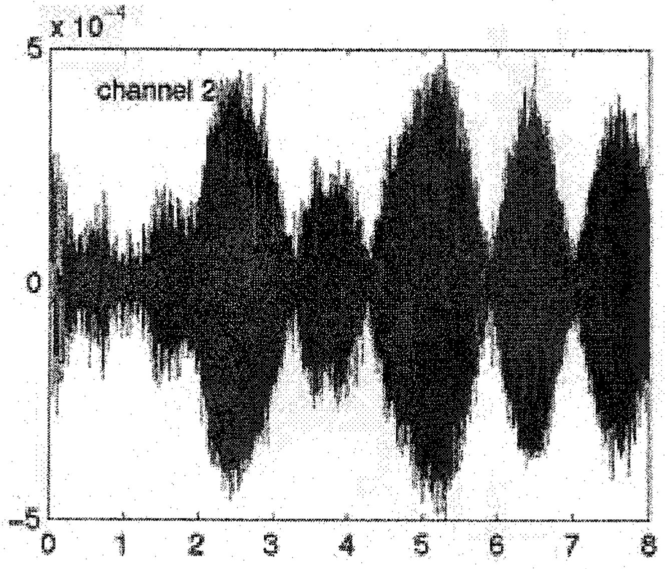


Figure 6.18 Quantization noise of MDLNS filterbank channel 2 (coefficient 5 3 5 3, input data 5 5 5 5)

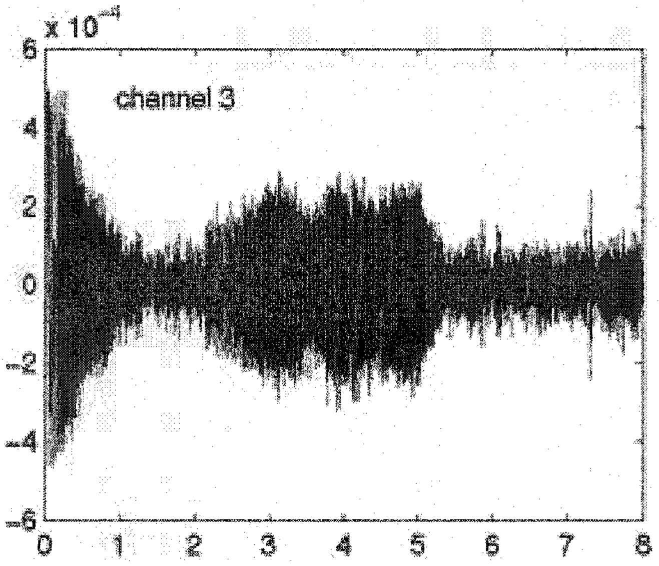


Figure 6.19 Quantization noise of MDLNS filterbank channel 3 (coefficient 5 3 5 3, input data 5 5 5 5)

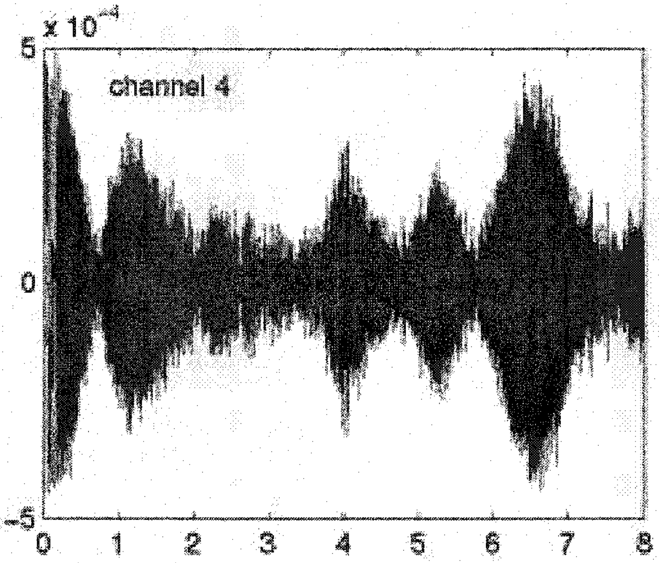


Figure 6.20 Quantization noise of MDLNS filterbank channel 4 (coefficient 5 3 5 3, input data 5 5 5 5)

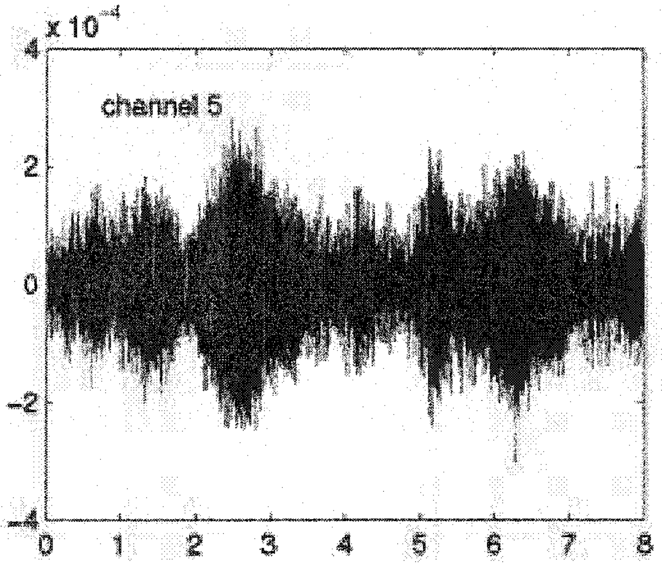


Figure 6.21 Quantization noise of MDLNS filterbank channel 5 (coefficient 5 3 5 3, input data 5 5 5 5)

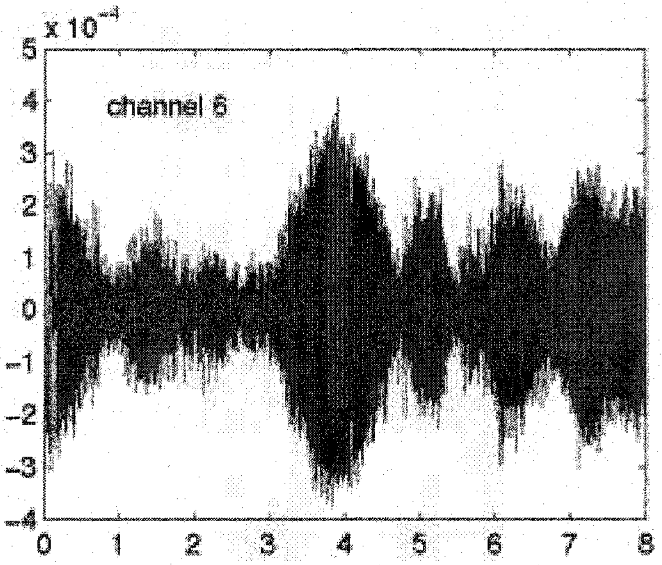


Figure 6.22 Quantization noise of MDLNS filterbank channel 6 (coefficient 5 3 5 3, input data 5 5 5 5)

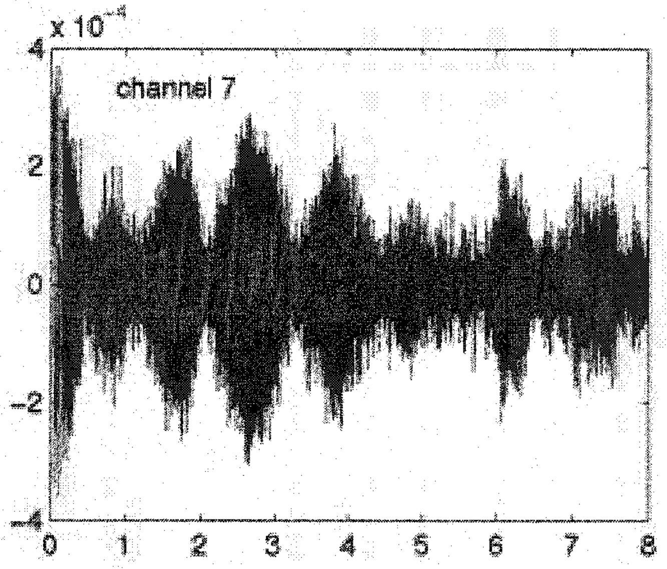


Figure 6.23 Quantization noise of MDLNS filterbank channel 7 (coefficient 5 3 5 3, input data 5 5 5 5)

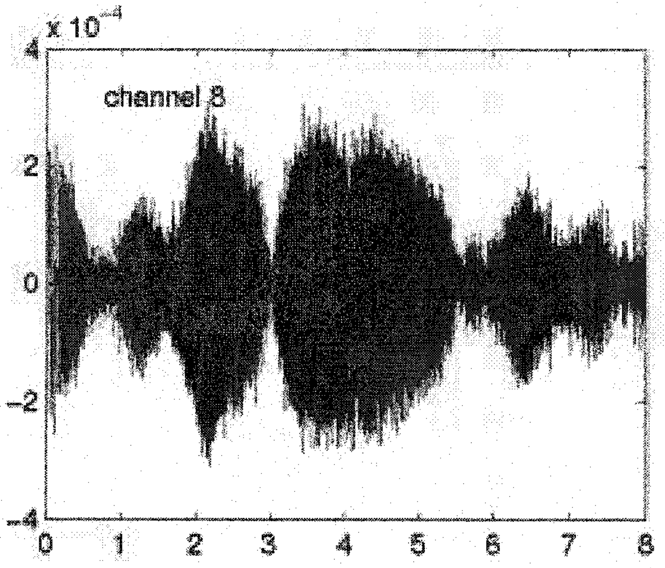


Figure 6.24 Quantization noise of MDLNS filterbank channel 8 (coefficient 5 3 5 3, input data 5 5 5 5)

As the figures show, the quantization noise range of the (5 5 5 5) MDLNS, input data implementation is $\pm 6 \times 10^{-4}$.

We see that the quantization noise of the (5 4 5 4) input data implementation is similar to the 16-bit binary implementation; the (5 5 5 5) input data implementation is better than the 16-bit binary implementation.

A negative effect is the signal modulation of the noise, that appears to be due to the non-linear representation of the MDLNS. Tests will have to be performed with human subjects in order to determine any possible deleterious effects this effect may have on the quality of the output signal.

6.3 Area Comparison

For comparison purposes we look at two recently published designs. A 16-bank filter, with a 60 dB stopband attenuation, using an DFT approach [24] and a 7-bank filter with a 40dB stopband attenuation [14].

The IFIR filterbank chip used in our comparison used a 0.7 μm CMOS technology. The die micrograph of this chip is shown in Figure 6.25. The layout of this design was provided by Oticon Inc. The layout was generated automatically using standard cells and a single-port RAM generator. From the chip micrograph in Figure 6.25, it is seen that all logic is placed in one block of standard cells at the bottom of the chip and that the RAM at the top of the chip has been divided into four blocks. Consequently, several IFIR filters are mapped onto each of the four RAM's. The chip contains 48 000 transistors and the size of the core is 3.6 x 2.7 mm (excluding pad cells). The transistors in the standard cells are scaled individually with small transistors for logic and larger transistors in the output drivers.

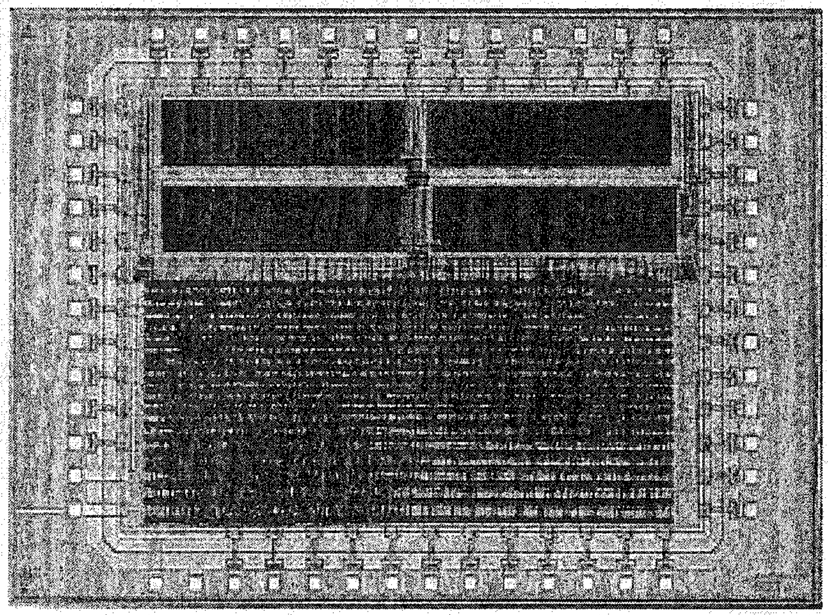


Figure 6.25 micrograph of the IFIR chip

The entire system of the DFT WOLA filterbank processor (including filterbank and I/O blocks) fabricated using 0.18 micron technology, requires an area of about 10 mm^2 [25].

The area of the MDLNS filterbank is 920 x 500 m^2, as mentioned in last chapter, which has the smallest area among these three filterbanks.

6.4 Power Comparison

Table 6.1 shows the Synopsys Power Compiler result for the MDLNS filterbank processor.

The total power of the filterbank is 0.626 mW and consumes 80.3% of the power requirements of the entire MDLNS filterbank processor chip.

Table 6.1 Synospys Power Compiler results of MDLNS filterbank processor

Hier-archy	Switch Power (mW)	Cell Power (mW)	Leak Power (pW)	Total Power (mW)	%
chip	0.238	0.519	2.31e7	0.780	100
ser.-par.	1.51e-4	8.97e-3	1.50e5	9.27e-3	1.2
convert	1.62e-3	7.26e-2	7.20e6	8.15e-2	10.4
filter	0.182	0.429	1.57e7	0.626	80.3
par.-ser.	1.34e-4	8.8e-3	8.99e4	2.23e-2	2.9

The DFT filterbank has a power consumption of 2.07 mW at 1.6 v. The 7-bank IFIR filter has a power consumption of 471 uW at 1.55v. Our design appears competitive, but it is important to point out that the design presented here represents an initial attempt at using the MDLNS technique, and there are many optimization procedures that we will be using in future designs that should considerably reduce the power consumption.

Chapter 7
Conclusion and Future Works

7.1 Conclusion

This thesis has explored the application of the Multidimensional Logarithmic Number System for a digital hearing aid filterbank implementation. An MDLNS filterbank processor chip has been designed and fabricated using 0.18 micron CMOS technology. The power and size of the filterbank test portion of the chip are 0.626 mW and 920 x 500 m^2 respectively.

We use an index calculus implementation of the MDLNS to take advantage of the logarithmic-like properties of the associated arithmetic. Using the index calculus form of the MDLNS, instead of the classical binary representation, a number of advantages are gained. The logarithmic-like properties of the index calculus MDLNS allow for reduced complexity multiplication and division, expensive operations using binary arithmetic. The multidimensional and multi-digit extensions of the representation reduce the complexity of computations compared to the Logarithmic Number System. Finally, the non-linear nature of the representation allows for a natural mapping of data and filter coefficients for the hearing instrument.

Comparing 1-digit and 2-digit MDLNS implementations, this new 2-digit MDLNS filterbank produces a much smaller second base coefficient. The size of the lookup table ROM is reduced by more than an order of magnitude compare to the single-digit implementation.

For hearing aid applications, the size, power, linear phase, and flat overall magnitude response are important constraints for the filterbank design. We have discovered that the MDLNS offers significant advantages over the standard binary system, mainly through overhead reduction in area achieved by not using multipliers. The MDLNS filterbank has linear phase with a perfectly flat overall magnitude response; a considerable improvement over the IFIR and DFT filterbank designs also presented in this thesis. The power and performance of the MDLNS filterbank are also competitive with the IFIR and DFT binary implementations.

7.2 Contributions

A novel algorithm and architecture have been developed for a 2-digit MDLNS filterbank, The final design resulted from an extensive study of two classical binary digital filterbanks: the interpolated FIR (IFIR) filterbank and the DFT filterbank. A comparison between these three forms of filterbank has been carried out. The successful design of this chip has shown that the MDLNS filterbank can be a viable design for digital hearing instruments, and are a practical alternative to binary implementations.

The MDLNS filterbank was integrated with the work of my fellow student (Roberto Muscedere), on novel conversion architectures for binary to MDLNS converters, and a complete MDLNS filterbank processor microchip was designed, simulated and fabricated as a proof of concept for our MDLNS filterbank architecture.

7.3 Suggestions for Future Work

The Virage RAM compiler has recently been made available to our laboratory, and so we will be able to generate the RAM with the exact size we need and with a single port instead of dual port design. This will save considerable power dissipation and area.

The MDLNS filterbank processor needs to be produced using custom cells and ROMS, This will demonstrate the true benefit from overhead reduction available by using the MDLNS.

If the Greedy algorithm or the modified Greedy Algorithm is used to generate both the data and the coefficients, then the first digit is selected as close as possible to the target number, with the second digit correcting for the error. This implies that the channel corresponding to the multiplication of the two low-order digits can be removed with acceptable precision reduction [10]. Only three channels need to be used instead of four channels with this approach.

In CMOS circuits, the power consumption is mainly related to signal transitions and stems from the charging and discharging of the parasitic capacitances in transistors and wires and from short-circuit currents during switching. Minimizing power consumption is therefore a question of avoiding unnecessary signal transitions which do not contribute to the computation in question. In synchronous design this is addressed by stopping the clock signal in unused modules. This is called clock gating and is only manageable at a coarse grain level. Asynchronous circuits have the inherent property of only activating modules and storage elements where and when needed. This may be viewed as a systematic way of introducing fine grain clock gating and variable length clocks Asynchronous implementation of the IFIR filterbank has shown to provide a five-fold power reduction compared to its synchronous counterpart [14]. We expect that asynchronous circuits, used to implement our MDLNS filterbank architecture, will provide similar levels of power saving.

REFERENCES

[1] Erkan Onat, “DSP algorithms for digital hearing instruments”, Master’s Thesis, University of Windsor, 2001.

[2] Sonic Innovations (2002). *Optimized Target Matching: Demonstration of an Adaptive Nonlinear DSP System*, [Online]. Available: http://www.sonici.com/images/advancedresearch/match.pdf.

[3] Oticon Inc. (2002), about hearing, [Online]. Available: http://www.oticon.com/eprise/main/Oticon/com/SEC_AboutHearing/LearnAboutHearing2/HearingAndHearingLoss/_Index

[4] University of Edinburgh, to hear and to listen, [Online]. Available: http://www.see.ed. ac.uk/~SLIg/epz/hearing.pdf

[5] V. S. Dimitrov, J. Eskritt, L. Imbert, G. A. Jullien and W.C. Miller, 2001, The use of the multi-dimensional logarithmic number system in DSP applications, Proceedings of the 15th IEEE Symposium on Computer Arithmetic, June, paper 151, 8-pages.

[6] V.S.Dimitrov, S.Sadeghi-Emamchaie, G.A.Jullien and W.C.Miller, A near canonical double-base number system with applications in DSP, *SPIE Conference on Signal Processing Algorithms*, vol. 2846, pp.14-25. 1996

[7] S.Sadeghi-Emamchaie, G.A. Jullien, V.S.Dimirtov and W.C.Miller, Digital arithmetic using cellular neural networks, Journal of Circuits, Systems and computers, No. 8, vol. 6, pp. 515-535, Dec. 1998.

[8] T.J. Sullivan, R.E. Morley, Jr., and G.L. Engel, “A VLSI FIR Digital Signal Processor Using Logarithmic Arithmetic”, in 1988 IEEE Workshop on VLSI Signal Processing, pp. 276-280, VLSI Signal Processing-III, IEEE Press, 1988.

[9] V. Dimitrov, G.A. Jullien, W.C. Miller, 1999, Theory and Applications of the Double-Base Number System, IEEE Trans. Computers, vol. 48, No. 10, Oct. 1999, pp.1098- 1107.

[10] Stanley Jonathan Eskritt, 2001, “Inner Product Computational Architectures Using the Double Base Number System”, M.A.Sc. thesis, University of Windsor.

[11] R. Muscedere, G.A. Jullien, V. Dimitrov, W.C. Miller, "Non-linear signal processing using index calculus DBNS arithmetic", Proc. SPIE Vol. 4116, Advanced Signal Processing Algorithms, Architectures, and Implementations X, Franklin T. Luk; Ed. pp. 247-257, 2000.

[12] W.J.Ellison, On a theorem of Sivasankaranayana, Seminars on number theory, 1970-1971, Technical Report No. 12, CNRS, Talene, 1971.

[13] T. Lunner and J. Hellgren, “A digital filterbank hearing aid - design, implementation and evaluation,” in *Proc.ICASSP’91,* Toronto, Ont., Canada, 1991, pp. 3661–3664.

[14] Lars s. Nielsen and Jens Spars, Designing Asynchronous Circuits for Low Power: An IFIR Filter Bank for a Digital Hearing Aid, Proceedings of the IEEE, VOL.87, NO.2 February 1999, pp.268-281.

[15] Crochiere, R.E. and Rabiner, L.R., Multirate Digital Signal Processing. Prentice-Hall Inc., 1983.

[16] Vaidyanathan, P.P., “Multirate Digital Filters, Filter Banks, Polyphase Networks, and Applications: A Tutorial,” Proc. IEEE, Vol. 78, No. 1, pp. 56-93, January 1990.

[17] Vaidyanathan, P.P., *Multirate Systems and Filter Bank*s. Prentice-Hall Inc., 1993.

[18] P. P. Vaidyanathan (1987), “Design and Implementation of Digital FIR Filters”, Handbook of Digital Signal Processing, Academic Press, San Diego CA, 1987.

[19] Vaidyanathan, P.P. and Nguyen, T.Q., Eigenfilters: A new Approach to Least-squares FIR Filter Design and Applications including Nyquist Filters, IEEE Trans. on Circuits and Systems, Vol. CAS-34, No. 1, January 1987, pp. 11-23.

[20] H. Li, G. A. Jullien, V. Dimitrov, M. Ahmadi, and W. C. Miller, “A 2-Digit Multidimensional Logarithmic Number System Filterbank for a Digital Hearing Aid Architecture”, pp. II 761-763, IEEE International Symposium on Circuits and Systems, May 26-29, 2002, Phoenix, Arizona.

[21] H. Li, R. Muscedere, V. Dimitrov and G. A. Jullien, “The Application of 2-D Logarithms to Low-power Hearing-aid Processor”, Vol. III, pp. 13-16, 45th IEEE International Midwest Symposium on Circuits and Systems, August 4-7, 2002, Tulsa, Oklahoma.

[22] Brennan, R.L., Schneider, T., “A Flexible Filterbank Structure for Extensive Signal Manipulations in Digital Hearing Aids,” Proc. ISCAS-98, Monterey, CA.

[23] Schneider, T., Brennan R.L., A Multichannel Compression Strategy for a Digital Hearing Aid, Proc. ICASSP-97, Munich, Germany, pp. 411-415.

[24] Schneider, T., Brennan R.L., “An Ultra Low-power Programmable DSP System for Hearing Aids and Other Audio Applications” ICSPAT’99 Proceeding, Nov 1-4, 1999, Orlando, FL.

[25] Schneider, T., Brennan R.L., “Embedded Ultra Low-Power Digital Signal Processing” Electronics, IEEE canadian review- summer 2000.

[26] R. Muscedere, V.S. Dimitrov, G.A. Jullien, and W.C. Miller, “Efficient Conversion From Binary to Multi-Digit Multi-Dimensional Logarithmic Number Systems using Arrays of Range Addressable Look-Up Tables”, 13th IEEE International Conference on Application-Specific Systems, Architecture and Processors (ASAP 2002),pp. 130-138.

[27] CMC (2003), Tutorial on Digital IC Test: Using the CMC Test Head & VXI Test System, [Online]. Available: http://www.cmc.ca/prod_serv/des_fab_test/support/digital_test/test_digital_tut.1.0.pdf

Appendix A
Matlab Code for Filterbanks

A.1 Introduction

In Chapter 3, the concept of both DFT and 8 parallel FIR filterbanks was introduced. The matlab code for designing these filterbanks are presented in this appendix.

A.2 Matlab Code for DFT Filterbank

A.2.1 Nyquist Eigenfilter

The Nyquist eigenfilter is prototype filter used in the DFT filterbank, The frequency response of the other banks are uniformly frequency shifted versions of this eigenfilter.

The impulse response of a Nyquist filter of length N satisfies

$$h_n = 0 \qquad \text{for } \left(n - \frac{N-1}{2}\right) = \text{a nonzero multiple of } K$$

These are also known as Kth-band filters. The cutoff frequencies are $\omega_p + \omega_s = 2\pi/K$, where K is the intersymbol duration.

The design of Nyquist filters requires an algorithm that incorporates both time-domain and frequency-domain constraints.

```
ORDER=64;
 ws=0.0895*pi;
 wp=0.0355*pi;
 a=0.5;
 bb=0.5;

 % find matrix P(n,m) according band edges ws and wp
for n=0:ORDER/2-1
for m=0:ORDER/2-1
         if n==m
                 p(n+1,m+1)=bb/pi*(1.5*wp-2/
(n+0.5)*sin((n+0.5)*wp)+1/(4*n+2)*sin((2*n+1)*wp))-a/
(2*pi)*(sin((2*n+1)*ws)/(2*n+1)-pi+ws);
         else    p(n+1,m+1)=bb/pi*(wp-1/
(n+0.5)*sin((n+0.5)*wp)-1/(m+0.5)*sin((m+0.5)*wp)+0.5/
(n+m+1)*sin((n+m+1)*wp)+0.5/(n-m)*sin((n-m)*wp))-a/
(2*pi)*(sin((n+m+1)*ws)/(n+m+1)+sin((n-m)*ws)/(n-m));
                         end
        end
end
save matrix_p p
% computer the eigenvector (filter coefficient) of a
real, symmetric, and positive-definite matrix P corre-
sponding to the smallest eigenvalue.
[v,d]=eig(p);
dd=diag(d);
OR=length(dd);

min_eigval=min(dd);
min_index=find(dd==min_eigval);
h_min1=v(:,min_index);
hh=fliplr(h_min1');
h_final=[hh h_min1'];
[h,w]=freqz(h_final,1,1024,16000);


% computer FIR filter using kaiser Window
f=[0 0.0625 0.0625 1];
m1=[1 1 0 0];
b1=fir2(ORDER-1,f,m1,kaiser(ORDER,1.83));
figure;
[kaiser_h,w]=freqz(b1,1,1024,16000);
plot(w,20*log10(abs(kaiser_h)),w,20*log10((abs(h)/
max(abs(h)))),'r')
```

```
title('Nyquist EigenFilter vs. Kaiser Window')
grid;
xlabel('Frequency - Hz - with Sampling rate of
16kHz');
ylabel('Magnitude - dB');
text(5000,-20,'Red: Eigen filter');
text(5000,-30,'Blue:Kaiser Window ');

% make Nyquist_eigenfilter: insert zero in every Kth
eigen filter coefficient except the mid-term of the
coefficients.
load matrix_p;
py=p;
py(17,:)=[];
py(:,17)=[];
[v1,d1]=eig(py);
dd1=diag(d1);
min_eigval1=min(dd1);
min_index1=find(dd1==min_eigval1);
b_min1=v1(:,min_index1);
b_min2=b_min1';
b_temp=[b_min2(1:16) 0 b_min2(17:31)];
b_final=[fliplr(b_temp) b_temp];
save Nyq_coef b_final;
[h1,w]=freqz(b_final,1,1024);
figure(1); hold on;
plot(w/pi,20*log10((abs(h)/max(abs(h1)))),'g');grid
on;
axis([0 0.5 -80 20]);
```

Matlab code of DFT filterbank simulation

```
load  Nyq_coef b_final; %load Nyquist eigenfilter
coefficients
k=16;
h_final=b_final;
h1=h_final(1:k);
h2=h_final(k+1:2*k);
h3=h_final(2*k+1:3*k);
h4=h_final(3*k+1:4*k);
input2=zeros(1,8);
input3=zeros(1,8);
input4=zeros(1,8);
input5=zeros(1,8);
input6=zeros(1,8);
input7=zeros(1,8);
```

```
input8=zeros(1,8);
f0=7800;
f2=340;
sf=20000;
M=8;

% input signal is cos signal with 7800 Hz and 340 Hz
for n=1:1024
   input(n)=cos(2*pi*n*f0/sf)+cos(2*pi*n*f2/sf);
end
figure(1);
y=(abs(fft(input,1024))); grid on;
f=sf*(0:511)/1024;
plot(f,y(1:512));
for m=1:10
   input1=input2;
   input2=input3;
   input3=input4;
   input4=input5;
   input5=input6;
   input6=input7;
   input7=input8;
   input8=input(1+M*(m-1):M*m);

   ym1=h1.*[input1 input2];
   ym2=h2.*[input3 input4];
   ym3=h3.*[input5 input6];
   ym4=h4.*[input7 input8];
   ytotal=ym1+ym2+ym3+ym4;

   for i=1:16
      ytotal1(i)=ytotal(mod((i-m*M),k)+1);
   end
   output=abs((fft(ytotal1)));
   figure(2);
     f1=sf*(0:7)/16;
   plot(f1+500,output(1:8),'*');grid on;
end
  xlabel('Frequency - Hz');
  ylabel('Magnitude - dB');
```

A.3 Eight 60 dB Parallel FIR Filters

All 8 FIR filter are using Kaiser window function. The first 7 filters have 74 taps and the last one has 75 taps.

```
%  ===== System data and specifications  =====  %
OVER = 400;
OVER2 = 400  ;
FS = 16000;
BAND_NUM = 8;
PASS_RIP = 0.01;        % in DB
STOP_ATT = 60;% in DB


orders = [];
BAND_WIDTH = FS/2/(BAND_NUM-1);
rip = 10^(-PASS_RIP/20);
att = 10^(-STOP_ATT/20);

%  ===== First Lowpass Filter Design =====  %
[N,Wn,BETA,TYPE] = kaiserord([BAND_WIDTH/2-
OVER2,BAND_WIDTH/2+OVER],[1 0],[rip att],FS);
b1 = fir1(N,Wn,TYPE,kaiser(N+1,BETA),'noscale');
save b1.txt b1 -ascii -tabs
save b1
orders=[orders;N];

%  ==== Bandpass filter design ====  %
   ii=1;
   freq1 = ii*BAND_WIDTH-OVER;
   freq2 = ii*BAND_WIDTH+OVER2;
   freq3 = (ii+1)*BAND_WIDTH-OVER2;
   freq4 = (ii+1)*BAND_WIDTH+OVER;

   freq1=freq1-BAND_WIDTH/2;
   freq2=freq2-BAND_WIDTH/2;
   freq3=freq3-BAND_WIDTH/2;
   freq4=freq4-BAND_WIDTH/2;
  [N,Wn,BETA,TYPE] = kaiserord([freq1 freq2 freq3
freq4],[0 1 0],[att rip att],FS);
   b2 = fir1(N,Wn,TYPE,kaiser(N+1,BETA),'noscale');
   save b2.txt b2 -ascii -tabs
   save b2
   orders=[orders;N];
   ii=2;
   freq1 = ii*BAND_WIDTH-OVER;
   freq2 = ii*BAND_WIDTH+OVER2;
   freq3 = (ii+1)*BAND_WIDTH-OVER2;
   freq4 = (ii+1)*BAND_WIDTH+OVER;
   freq1=freq1-BAND_WIDTH/2;
   freq2=freq2-BAND_WIDTH/2;
   freq3=freq3-BAND_WIDTH/2;
   freq4=freq4-BAND_WIDTH/2;
  [N,Wn,BETA,TYPE] = kaiserord([freq1 freq2 freq3
freq4],[0 1 0],[att rip att],FS);
   b3 = fir1(N,Wn,TYPE,kaiser(N+1,BETA),'noscale');
   save b3.txt b3 -ascii -tabs
   save b3
   orders=[orders;N];
   ii=3;
   freq1 = ii*BAND_WIDTH-OVER;
   freq2 = ii*BAND_WIDTH+OVER2;
   freq3 = (ii+1)*BAND_WIDTH-OVER2;
   freq4 = (ii+1)*BAND_WIDTH+OVER;
   freq1=freq1-BAND_WIDTH/2;
   freq2=freq2-BAND_WIDTH/2;
   freq3=freq3-BAND_WIDTH/2;
   freq4=freq4-BAND_WIDTH/2;
   [N,Wn,BETA,TYPE] = kaiserord([freq1 freq2 freq3
freq4],[0 1 0],[att rip att],FS);
   b4 = fir1(N,Wn,TYPE,kaiser(N+1,BETA),'noscale');
   save b4.txt b4 -ascii -tabs
   save b4
   orders=[orders;N];
   ii=4;
   freq1 = ii*BAND_WIDTH-OVER;
   freq2 = ii*BAND_WIDTH+OVER2;
   freq3 = (ii+1)*BAND_WIDTH-OVER2;
   freq4 = (ii+1)*BAND_WIDTH+OVER;
   freq1=freq1-BAND_WIDTH/2;
   freq2=freq2-BAND_WIDTH/2;
   freq3=freq3-BAND_WIDTH/2;
   freq4=freq4-BAND_WIDTH/2;
   [N,Wn,BETA,TYPE] = kaiserord([freq1 freq2 freq3
freq4],[0 1 0],[att rip att],FS);
   b5 = fir1(N,Wn,TYPE,kaiser(N+1,BETA),'noscale');
   save b5.txt b5 -ascii -tabs
   save b5
   orders=[orders;N];
   ii=5;
   freq1 = ii*BAND_WIDTH-OVER;
   freq2 = ii*BAND_WIDTH+OVER2;
   freq3 = (ii+1)*BAND_WIDTH-OVER2;
   freq4 = (ii+1)*BAND_WIDTH+OVER;
   freq1=freq1-BAND_WIDTH/2;
   freq2=freq2-BAND_WIDTH/2;
   freq3=freq3-BAND_WIDTH/2;
   freq4=freq4-BAND_WIDTH/2;
   [N,Wn,BETA,TYPE] = kaiserord([freq1 freq2 freq3
freq4],[0 1 0],[att rip att],FS);
```

```
   b6 = fir1(N,Wn,TYPE,kaiser(N+1,BETA),'noscale');
   save b6.txt b6 -ascii -tabs
   save b6
   orders=[orders;N];
   ii=6;
   freq1 = ii*BAND_WIDTH-OVER;
   freq2 = ii*BAND_WIDTH+OVER2;
   freq3 = (ii+1)*BAND_WIDTH-OVER2;
   freq4 = (ii+1)*BAND_WIDTH+OVER;
   freq1=freq1-BAND_WIDTH/2;
   freq2=freq2-BAND_WIDTH/2;
   freq3=freq3-BAND_WIDTH/2;
   freq4=freq4-BAND_WIDTH/2;
   [N,Wn,BETA,TYPE] = kaiserord([freq1 freq2 freq3
freq4],[0 1 0],[att rip att],FS);
   b7 = fir1(N,Wn,TYPE,kaiser(N+1,BETA),'noscale');
   save b7.txt b7 -ascii -tabs
   save b7
   orders=[orders;N];
   [N,Wn,BETA,TYPE] = kaiserord([FS/2-BAND_WIDTH/2-
OVER,FS/2-BAND_WIDTH/2+OVER2],[0 1],[att rip],FS);
   b8 = fir1(N,Wn,TYPE,kaiser(N+1,BETA),'noscale');
   save b8.txt b8 -ascii -tabs
   save b8
   orders=[orders;N];
   save orders.txt orders -ascii -tabs
   %  =====Freq response for all the FIR filters
======   %

   [H1,F] = freqz(b1,1,2000,FS);
   [H2,F] = freqz(b2,1,2000,FS);
   [H3,F] = freqz(b3,1,2000,FS);
   [H4,F] = freqz(b4,1,2000,FS);
   [H5,F] = freqz(b5,1,2000,FS);
   [H6,F] = freqz(b6,1,2000,FS);
   [H7,F] = freqz(b7,1,2000,FS);
   [H8,F] = freqz(b8,1,2000,FS);
H=H1+H2+H3+H4+H5+H6+H7+H8;
figure;

plot(F,20*log10(abs(H1)),'y',F,20*log10(abs(H2)),'m',
F,20*log10(abs(H3)),'c',F,20*log10(abs(H4)),'r',F,20*
log10(abs(H5)),'g',F,20*log10(abs(H6)),'b',F,20*log10
(abs(H7)),'k',F,20*log10(abs(H8)),'r');
ylabel('Magnitude(dB)')
xlabel('Frequency(Hz)')
title('Magnitude Responses of Passbands')
   axis([0 FS/2 -150 5])
   grid on; zoom on;
   figure;
   plot(F,20*log10(abs(H)))
   axis([0 FS/2 -25 10])
   ylabel('Magnitude(dB)')
    % ylabel('Magnitude')
% xlabel('Frequency(Hz)')
   title('Optimized Overall Magnitude Response')
   grid on; zoom on;
   figure;
   plot(F,unwrap(angle(H1))*180/
pi,'y',F,unwrap(angle(H2))*180/
pi,'m',F,unwrap(angle(H3))*180/
pi,'c',F,unwrap(angle(H4))*180/
pi,'r',F,unwrap(angle(H5))*180/
pi,'g',F,unwrap(angle(H6))*180/
pi,'b',F,unwrap(angle(H7))*180/
pi,'k',F,unwrap(angle(H8))*180/pi,'r');
   ylabel('Phase(degrees)')
   xlabel('Frequency(Hz)')
   grid on;  zoom on;
   orders
```

Improved 8 parallel 60 dB filterbank by Mr. Roberto Muscedere. All eight filters have 75 taps, and make 8 perfect symmetrical filters

```
%  clear all;
close all
clc;

%  ===== System data and specifications  =====  %
OVER = 400;
OVER2 = 400;
FS = 16000;
BAND_NUM = 8;

PASS_RIP = 0.01;% in DB
STOP_ATT = 60;% in DB

BAND_WIDTH = FS/2/(BAND_NUM-1);
rip = 10^(-PASS_RIP/20);
att = 10^(-STOP_ATT/20);

RES = 4096;
```

```
% Generate half of the filters
for i = 1:BAND_NUM,
if i==1
% For the first case, find the BETA and order of the
filters
   Wn = [BAND_WIDTH/2-OVER2 BAND_WIDTH/2+OVER];
   [N,Wn,BETA,TYPE] = kaiserord(Wn,[1 0],[rip
att],FS);
% If the order is odd, make it even
   if mod(N,2)==1
         N=N+1;
   end
% setup a storage array for all the coefficients
   b=zeros(BAND_NUM,N+1);
   disp(sprintf('Coefficients = %d',N));
elseif i==BAND_NUM
% For the last case
   Wn = [FS/2-BAND_WIDTH/2-OVER FS/2-BAND_WIDTH/
2+OVER2];
   [dummy,Wn,dummy2,TYPE] = kaiserord(Wn,[0 1],[att
rip],FS);
else
% Set up for other filters
   Wn = [-OVER OVER2 BAND_WIDTH-OVER2
BAND_WIDTH+OVER];
   Wn = Wn + BAND_WIDTH*(i-1) - BAND_WIDTH/2;
% Geneate the normalize frequency range and type
   [dummy,Wn,dummy2,TYPE] = kaiserord(Wn,[0 1 0],[att
rip att],FS);
end
% make the coefficients
b(i,:) = fir1(N,Wn,TYPE,kaiser(N+1,BETA),'noscale');
end

% Determine symmetry
if mod(ceil((N+1)/2),2)==1
disp('Odd symmetry.');
% Odd coefficients are negated
else
% Even coefficients are negated
disp('Even symmetry.');
end

cols='ymcrrcmy';

% Setup storage for the magnitude response
H = zeros(BAND_NUM,RES);
F = zeros(1,RES);
```

```
figure;

% Plot each response
for i = 1:BAND_NUM,
[dummy,F] = freqz(b(i,:),1,RES,FS);
H(i,:) = dummy';

plot(F,20*log10(abs(H(i,:))),cols(mod(i-1,length(cols))+1));
hold on;
end
hold off;

title('Magnitude Responses of Passbands')
ylabel('Magnitude (dB)')
xlabel('Frequency (Hz)')
axis([0 FS/2 -150 5])
grid on;
zoom on;

% plot the total magnitute response
figure;
T=sum(H,1);
plot(F,20*log10(abs(T)))
axis([0 FS/2 -25 10])
ylabel('Magnitude (dB)')
xlabel('Frequency (Hz)')
title('Optimized Overall Magnitude Response')
grid on;
zoom on;

% plot the phase response
figure;
for i = 1:BAND_NUM,
plot(F,unwrap(angle(H(i,:)))*180/pi,cols(mod(i-
1,length(cols))+1));
hold on;
end
hold off;
title('Phase Responses of Passbands')
ylabel('Phase (degrees)')
xlabel('Frequency (Hz)')
grid on;
zoom on;

for i = 1:BAND_NUM,
low = BAND_WIDTH*(i-1)-BAND_WIDTH/2-OVER;
```

```
low2 = BAND_WIDTH*(i-1)-BAND_WIDTH/2+OVER2;
high = BAND_WIDTH*(i-1)+BAND_WIDTH/2+OVER;
high2 = BAND_WIDTH*(i-1)+BAND_WIDTH/2-OVER2;
worst = -1e9;
worst2 = 0;
%disp(sprintf('low=%f high=%f',low,high));
for j=1:RES,
   dummy=20*log10(abs(H(i,j)));
   if (F(j)>=low2) & (F(j)<=high2)
         if (dummy<-PASS_RIP | dummy >PASS_RIP)
%              disp(sprintf('Filter %d @ %fHz =
%fdB',i,F(j),dummy));
         end
         if (abs(dummy)>abs(worst2))
               worst2=dummy;
         end
   end
   if (F(j)<=low) | (F(j)>=high)
         if (dummy>-STOP_ATT)
%              disp(sprintf('Filter %d @ %fHz =
%fdB',i,F(j),dummy));
         end
         if dummy>worst
               worst=dummy;
         end
   end
end
disp(sprintf('Filter %3d worst case %+fdB &
%+fdB',i,worst,worst2));
end

fid=fopen('coefs.in','w');
if fid>0
fwrite(fid,b','double');
fclose(fid);
else
disp('Cannot write coefs to file.');
break;
end

disp('Execute: ./optbase-doublebrute 2 coefs.in
coefs.out');
disp('Press a key when done optimizing...');
pause;

fid=fopen('coefs.out','r');
if fid>0
c=fread(fid,size(b'),'double');
```

```
c=c';
cb=fread(fid,1,'double');
ce=fread(fid,1,'double');
fclose(fid);
else
disp('Cannot read coefs from file.');
break;
end

% Setup storage for the magnitude response
H2 = zeros(BAND_NUM,RES);

figure;

% Plot each response
for i = 1:BAND_NUM,
[dummy,F] = freqz(c(i,:),1,RES,FS);
H2(i,:) = dummy';

plot(F,20*log10(abs(H2(i,:))),cols(mod(i-
1,length(cols))+1));
hold on;
end
hold off;
title('Magnitude Responses of Passbands (MDLNS)')
ylabel('Magnitude (dB)')
xlabel('Frequency (Hz)')
axis([0 FS/2 -150 5])
grid on;
zoom on;

% plot the total magnitute response
figure;
T2=sum(H2,1);
plot(F,20*log10(abs(T2)))
axis([0 FS/2 -25 10])
ylabel('Magnitude (dB)')
xlabel('Frequency (Hz)')
title('Optimized Overall Magnitude Response (MDLNS)')
grid on;
zoom on;

for i = 1:BAND_NUM,
low = BAND_WIDTH*(i-1)-BAND_WIDTH/2-OVER;
low2 = BAND_WIDTH*(i-1)-BAND_WIDTH/2+OVER2;
high = BAND_WIDTH*(i-1)+BAND_WIDTH/2+OVER;
high2 = BAND_WIDTH*(i-1)+BAND_WIDTH/2-OVER2;
```

```
worst = -1e9;
worst2 = 0;
%disp(sprintf('low=%f high=%f',low,high));
for j=1:RES,
   dummy=20*log10(abs(H2(i,j)));
   if (F(j)>=low2) & (F(j)<=high2)
         if (dummy<-PASS_RIP | dummy >PASS_RIP)
%              disp(sprintf('Filter %d @ %fHz =
%fdB',i,F(j),dummy));
         end
         if (abs(dummy)>abs(worst2))
              worst2=dummy;
         end
   end
   if (F(j)<=low) | (F(j)>=high)
         if (dummy>-STOP_ATT)
%              disp(sprintf('Filter %d @ %fHz =
%fdB',i,F(j),dummy));
         end
         if dummy>worst
              worst=dummy;
         end
   end
end
disp(sprintf('Filter %3d worst case %+fdB &
%+fdB',i,worst,worst2));
end
```

152 unique coefficients using a 2-digit MDLNS representation

```
1     -17   1     1     -12   -2
1     -16   0     1     -12   0
1     -11   -1    -1    -18   -1
1     -11   0     1     -16   -1
1     -10   -2    1     -16   0
1     -10   0     -1    -11   -2
1     -10   -1    -1    -14   1
1     -10   -2    -1    -18   -1
1     -11   -1    -1    -17   1
0     0     0     0     0     0
-1    -8    -1    1     -9    1
-1    -14   0     -1    -9    -2
-1    -9    -1    -1    -10   -2
-1    -8    -1    -1    -14   0
-1    -7    -2    1     -10   -2
-1    -11   0     -1    -7    -2
-1    -8    1     -1    -9    -2
```

```
-1     -8     -1     -1     -8     0
-1     -7     -1     -1     -10    1
-1     -7     -1     -1     -9     -2
-1     -7     0      1      -9     -2
-1     -11    0      -1     -7     -2
-1     -8     -1     -1     -18    -1
0      0      0      0      0      0
1      -7     1      -1     -7     -1
1      -5     -1     -1     -6     0
1      -5     -1     -1     -7     1
1      -6     1      1      -11    0
1      -5     0      -1     -8     0
1      -5     -1     1      -7     1
1      -5     1      1      -10    1
1      -4     -1     -1     -11    -1
1      -4     0      -1     -7     0
1      -3     -2     -1     -6     0
1      -4     0      1      -9     1
1      -7     -1     1      -4     0
1      -4     1      -1     -6     -2
1      -5     0      1      -5     1
-1     -12    0      1      -14    -1
-1     -11    0      1      -16    1
-1     -10    -1     1      -17    -1
-1     -10    0      1      -13    -2
-1     -10    -1     1      -19    0
-1     -11    -1     1      -14    1
1      -11    -2     1      -17    0
1      -10    -1     -1     -15    -1
1      -10    -2     1      -13    -2
0      0      0      0      0      0
-1     -10    0      1      -17    -1
-1     -9     -1     -1     -15    0
-1     -9     -2     1      -12    0
1      -11    1      1      -10    -1
1      -8     1      1      -12    -1
1      -6     -2     -1     -14    0
1      -7     1      1      -8     -2
1      -6     -1     1      -12    1
1      -8     -1     1      -7     -1
1      -12    -1     1      -8     -1
-1     -8     -1     1      -13    -1
-1     -8     1      -1     -9     -1
-1     -7     -1     1      -10    -2
0      0      0      0      0      0
1      -10    0      1      -7     -1
1      -6     0      -1     -7     -2
1      -11    -2     1      -7     -1
-1     -6     -2     1      -12    1
-1     -4     -2     1      -8     0
-1     -4     0      1      -10    -2
-1     -4     1      -1     -9     1
-1     -3     0      1      -4     -2
-1     0      1      1      1      -2
```

```
-1      -5      -1      -1      -9      1
1       -7      1       1       -5      -2
1       -3      -2      1       -6      -2
1       -3      0       1       -8      -2
1       -4      0       1       -4      1
-1      -13     -2      1       -18     0
1       -12     1       1       -17     1
1       -10     -1      -1      -17     -1
1       -11     1       -1      -18     -1
-1      -13     0       -1      -12     -2
-1      -9      -2      -1      -14     -2
-1      -10     1       1       -15     -1
-1      -12     0       -1      -15     -2
1       -12     -2      1       -12     1
0       0       0       0       0       0
-1      -14     1       -1      -10     -2
1       -10     -2      -1      -14     -2
1       -8      0       -1      -13     -1
1       -7      -1      -1      -11     0
1       -9      0       -1      -13     -1
-1      -8      1       -1      -9      -1
-1      -7      1       -1      -8      -2
-1      -8      0       -1      -7      -2
1       -9      1       1       -10     -1
1       -6      0       -1      -9      1
1       -6      -1      -1      -11     -2
1       -8      -2      -1      -14     -1
-1      -8      -1      -1      -10     -1
0       0       0       0       0       0
1       -7      -1      -1      -9      -2
-1      -8      0       1       -16     1
-1      -4      -2      1       -6      -1
-1      -5      1       1       -8      -1
-1      -6      -1      1       -16     0
1       -8      1       1       -4      -2
1       -4      1       1       -9      1
1       -3      -2      -1      -6      0
-1      -5      -1      1       -21     1
-1      -1      -2      1       -2      -1
-1      -3      0       1       -7      0
-1      -5      0       1       -10     -1
1       -3      -2      1       -6      -1
1       -4      0       1       -4      1
1       -11     -2      -1      -16     -1
-1      -14     1       -1      -14     -2
-1      -10     -1      1       -17     -1
-1      -12     -2      -1      -13     -2
1       -10     1       -1      -13     1
1       -8      -2      -1      -9      -1
-1      -8      -2      1       -9      -1
-1      -9      -2      1       -13     0
1       -12     1       -1      -12     -2
0       0       0       0       0       0
-1      -12     0       1       -17     -2
```

```
1       -9      0       1       -11     -2
1       -8      1       -1      -8      -2
-1      -8      0       1       -15     0
-1      -8      1       -1      -9      1
1       -8      -2      -1      -14     -1
1       -7      1       1       -8      -2
1       -8      -1      1       -15     1
-1      -6      0       1       -8      -2
-1      -7      0       -1      -10     1
1       -6      -2      -1      -10     1
1       -6      -2      -1      -14     0
-1      -10     0       -1      -11     -1
0       0       0       0       0       0
1       -8      -2      -1      -11     1
-1      -5      1       1       -5      -1
-1      -7      0       -1      -7      1
1       -7      1       1       -6      0
1       -4      -1      1       -11     0
-1      -6      1       1       -7      -2
-1      -4      1       -1      -9      1
-1      -3      -1      1       -3      -2
1       -3      -1      1       -9      -2
1       -3      -2      -1      -11     1
-1      -4      1       -1      -10     0
-1      -3      0       1       -9      -1
1       -4      1       -1      -4      -1
1       -4      0       1       -4      1
```

Appendix B

Verilog Code for MDLNS Filterbank

B.1 Testbench.v

//Testbench for entire MDLNS filterbank system.

```
`timescale 1ns/10ps

            //Define main module
module topfilter;
            //include parameters
`include "parameters.v"

reg CK,reset,enable,inputenable;
integer count,count1,n;
reg [2:0] col;
reg inputbit;

      //Set up an array for input
reg[dsize:0] binaryinputs[0:numinputs-1],binaryintemp;
      //set up array for output
reg[aosize-2:0] a,b,c,d,e,f,g,h,outtop1,outtop2,
                outtop_tmp1, outtop_tmp2;
wire start,ready;

      //instantiate MDLNS filterbank
chipfilterbank_convert
#(dbsize,dtsize,cbsize,ctsize,aosize,mbsize,lbsize,lt
size,absize,sbsize,aisize,mtsize,num-
coef,dsize,wsize,isize,ssize,bsize,tsize,csize,mdiff,
maxlevel) filterchip(CK,reset,enable,input-
bit,sout1,sout2,ready,start);

initial
begin
            //read input chirp signal from a table
      $readmemb("inputchirp9000.lut",binaryinputs,0,numinputs-1);
      binaryintemp=binaryinputs[0];
                   //initialize
      CK=1'b0;
      n=0;
      col=3'b0;
      reset=1'b0;
      count=0;
      count1=0;
      enable=0;
      inputenable=0;
      outtop1 = 17'b0;
      outtop_tmp1 = 17'b0;
      outtop2 = 17'b0;
      outtop_tmp2 = 17'b0;
      #200 reset=1'b1;
end


                //setup clock to pulse every 100 units
always #100 CK=!CK;

always #62600   //every 313 clock cycles send an input data
begin
      binaryintemp=binaryinputs[n];
      n=n+1;
      inputenable=1;
end

always @(posedge CK) //convert parallel input to serial input
begin
      if (inputenable && count1<16)
         begin
           enable=1;
           inputbit=binaryintemp[count1];
           count1=count1+1;
         end

      else if (count1==16)
         begin
           enable=0;
           inputenable=0;
           count1=0;
         end

end

always @(posedge CK)  //convert output from serial to parallel in order
begin                 //to display filterbank output result

      if (ready && count<=16)
         begin
           outtop_tmp1[count]=sout1;
           outtop_tmp2[count]=sout2;
```

```
            count=count+1;
          end

      else if (count==17)
          begin
            outtop1 = outtop_tmp1;
            outtop2 = outtop_tmp2;
            count = 0;
            if (col==3'd0) begin a=outtop1; h=outtop2; end
            else  if (col==3'd1) begin b=outtop1;  g=outtop2; end
            else  if (col==3'd2) begin c=outtop1;  f=outtop2; end
            else
                   begin
                      d=outtop1; e=outtop2;
                      col=-1;
                   end
            $display ("+%b%b%b%b%b%b%b%b",a,b,c,d,e,f,g,h);
            col = col + 1;
         end
end

            //total 18,500 input data
always @(n)
      if(n==18500)  $finish;

endmodule
```

B.2 parameters.v

The parameter file includes all the variable parameters, such as bit width and bus size. These parameters are stored here for global use in all Verilog throughout the design.

```
parameter ctsize =1,cbsize=5, dbsize=5, dtsize=4;
parameter mbsize=6, mtsize=4;
parameter lbsize=5, ltsize=16, absize=7, sbsize=15;
parameter aisize=18,aosize=18;
parameter dsize=15;
parameter wsize=16,isize=17,ssize=5,bsize=6,tsize=5,
          csize=2,mdiff=11,maxlevel=1;
parameter numcoef=152;
parameter numinputs=18500;
```

B.3 chipfilterbank.v

Top level of MDLNS filterbank chip which has four parts : input block (serial to parallel converter); binary to MDLNS converter; MDLNS filterbank and control unit block; output block (parallel to serial converter).

```
module chipfilterbank(CK,reset,enable,inputbit,sout1,sout2,ready,start);

// Default parameters
parameter
dbsize=5,dtsize=4,cbsize=5,ctsize=1,aosize=18,mbsize=6,lbsize=5,ltsize=
16,absize=7,sbsize=15,aisize=18,mtsize=4,numcoef=152,dsize=15;
parameter wsize=16,isize=17,ssize=5,bsize=6,tsize=5,
csize=2,mdiff=11,maxlevel=1;

// Define ports
input       CK, reset, inputbit,enable;
output      ready;
output      sout1,sout2,start;
wire [wsize-1:0] i0;
wire [1:0]        datasign1,datasign2;
wire [dbsize:0]   databin1,databin2;
wire [dtsize:0]   datatern1,datatern2;
wire              activate,ready1;
wire [dsize:0]    binaryin;
wire [aosize-2:0] out1,out2;

//instantiate input serial to parallel block
s2p #(dsize)  s2p1
      (
      .datain           (inputbit),
      .CK               (CK),
      .enable           (enable),
      .dataout          (binaryin[dsize:0]),
      .reset            (reset)
      );

//instantiate the binary to MDLNS convert block
top_convert  #(wsize, isize,ssize,bsize,tsize,csize,mdiff,maxlevel)
serial(
      .CK               (CK),
      .reset            (!reset),
      .activate         (activate),
      .i0               (binaryin[dsize:0]),
      .ready            (ready1),
      .output_s         ({datasign2[1:0],datasign1[1:0]}),
      .output_b         ({databin2[dbsize:0],databin1[dbsize:0]}),
      .output_t         ({datatern2[dtsize:0],datatern1[dtsize:0]})
      );
```

```
//instantiate the MDLNS filterbank and control unit block
controlfilter
#(dbsize,dtsize,cbsize,ctsize,aosize,mbsize,lbsize,ltsize,absize,sbsize
,aisize,mtsize,numcoef)
controlunit(
      .CK                 (CK),
      .reset              (reset),
      .datasign1          (datasign1[1:0]),
      .databin1           (databin1[dbsize:0]),
      .datatern1          (datatern1[dtsize:0]),
      .datasign2          (datasign2[1:0]),
      .databin2           (databin2[dbsize:0]),
      .datatern2          (datatern2[dtsize:0]),
      .out1               (out1[aosize-2:0]),
      .out2               (out2[aosize-2:0]),
      .wout               (wout),
      .activate           (activate)
      );

//instantiate the output parallel to serial converter block
p2s #(aosize) p2s1
      (
      .pinput1            (out1[aosize-2:0]),
      .pinput2            (out2[aosize-2:0]),
      .sout1              (sout1),
      .sout2              (sout2),
      .CK                 (CK),
      .reset              (reset),
      .en                 (wout),
      .ready              (ready),
      .start              (start)
       );

endmodule
```

B.3.1 s2p.v

```
module s2p(datain,CK,reset,enable,dataout);

      // Default parameter;
parameter dsize=15;

      //Input & output Ports;
input datain,CK,enable,reset;
output [dsize:0] dataout;

reg [dsize:0] dataout;
reg  [4:0] count;

always @(posedge CK)
   if (!reset)
     begin
       dataout=15'b0;
```

```
          count = 5'd0;
        end
      else if (enable && count<5'd16)
            begin
              dataout[count]=datain;
              count = count+1'd1;
            end
          else   count = 5'd0;

endmodule
```

B.3.2 controlfilter.v

```
//Define MDLNS filterbank and control unit

module controlfilter
(CK,reset,datasign1,databin1,datatern1,datasign2,databin2,datatern2,out
1,out2,wout,activate);

      // Default parameters
parameter
dbsize=2,dtsize=4,cbsize=2,ctsize=4,aosize=18,mbsize=2,lbsize=1,ltsize=
1,absize=1,sbsize=1,aisize=1,mtsize=1,numcoef=1;

      // Define Ports
  input                   CK,reset;
  input  [1:0]            datasign1,datasign2;
  input  [dbsize:0]       databin1,databin2;
  input  [dtsize:0]       datatern1,datatern2;
  output [aosize-2:0]     out1,out2;
  output                  wout,activate;

      // Define control signals
  reg                     clear1,oew,wea;
  reg                     wout,activate,coefmiddle;

      //Define coefficient registers(will be read in);
  reg [1:0]               coefsignrom1[0:numcoef-1];
  reg [cbsize:0]          coefbinrom1[0:numcoef-1];
  reg [ctsize:0]          coefternrom1[0:numcoef-1];
  reg [1:0]               coefsignrom2[0:numcoef-1];
  reg [cbsize:0]          coefbinrom2[0:numcoef-1];
  reg [ctsize:0]          coefternrom2[0:numcoef-1];
  reg [1:0]               coefsign1,coefsign2;
  reg [cbsize:0]          coefbin1,coefbin2;
  reg [ctsize:0]          coeftern1,coeftern2;

      // Define input data
  wire [1:0]              datasign$1,datasign$2;
  wire [dbsize:0]         databin$1,databin$2;
  wire [dtsize:0]         datatern$1,datatern$2;
  wire [5:0]              datazero;
```

```
      // Define counters
  reg  [8:0]            k;
  reg[7:0]              j;
  reg  [6:0]            b,p;
  reg  [7:0]            adra,adrb;

      // instantiate the MDLNS filterbank block
dbnsfilterchip_ram
#(aosize,dbsize,dtsize,cbsize,ctsize,mbsize,mtsize,lbsize,ltsize,absize
,sbsize,aisize) filter1(out1,out2, CK, coefsign1, coefsign2,coefbin1,
coefbin2, coeftern1, coeftern2, datasign$1, datasign$2,databin$1, dat-
abin$2 , datatern$1, datatern$2,clear1,reset,wout,j[0] );

      // instantiate the RAM to hold the input data
rfss2_256x32cm4 ram1
(.QB({datazero,datasign$1,daabin$1,datatern$1,datasign$2,dat-
abin$2,datatern$2}) , .ADRA(adra) ,
.DA({6'b0,datasign1,databin1,datatern1,datasign2,databin2,datatern2}) ,
.WEA(wea) , .MEA(1'b1) , .CLKA(!CK) , .ADRB(adrb) , .OEB(oew) ,
.MEB(1'b1) , .CLKB(!CK)  );

      //initialzation routine
always @(reset)
begin
      // making coefficient ROM
  'include "coefrom152.v"
end

always @(posedge CK)
begin
      if (!reset)
        begin
           clear1=1'b0;
           j=0;    //counter for coefficient ROM address
           k=0;    // counter for processing one input data
           p=0;   // counter for input data address
           coefmiddle=0; // middle point flag of filter coefficients
           activate=0;
           wout = 0;

           adra=0;       //write address of RAM
           adrb=0;       //read address of RAM
           oew=0;        // RAM read enable
           wea=0;        // RAM write enable
        end

      else
        begin
          if (k==0)
             begin
               wea=1;
                adra=p;
                p=p+1;
                if (p==75) p=0;
          b=adra;
       end

            //after finishing calculate each FIR filter
    else if ((k==76) || (k==154) || (k==232) || (k==310) )
      begin
             // input 0 for accumulator
        coefsign1=2'b00;
        coefsign2=2'b00;
             // reset the middle point flag of a filter coefficient
         coefmiddle=0;
             // counter point to first coefficient of next filter
        j=j+39;
            // disable output of RAM
        oew=0;
      end
            //enable output of each filter when
            //finishing computation of a filter

    else if ((k == 77) || (k==155) || (k==233) || (k==311) )
       begin
         wout = 1;
       end

            // then reset output enable signal
            // and clear output register
    else if ((k == 78) || (k==156) || (k==234) || (k==312))
       begin
         wout = 0;
         clear1 = 1'b0;
            //reset counter when finish computation
            // of all 8 filters
         if (k==312)
            begin
              k=-1;
              j=0;
            end
       end

else
  begin
            // set the coefficient middle point flag
            // due to the symmetrical of the filter coefficient
    if ((k==38) || (k==116) || (k==194) ||(k==272))
      coefmiddle=1;

            // start to calculate each filter
    if ((k==1) || (k == 79) || (k==157) || (k==235) )
      begin
        clear1 = 1'b1;
        wea=0;         // disable write to RAM
        oew=1'b1;      // enable read from RAM
      end
```

```
          adrb=b;              // read input data

                  // read coefficients
          coefsign1=coefsignrom1[j];
          coefsign2=coefsignrom2[j];
          coefbin1=coefbinrom1[j];
          coefbin2=coefbinrom2[j];
          coeftern1=coefternrom1[j];
          coeftern2=coefternrom2[j];
                  // read previous coefficients after
                  //computate the middle coefficient of each filter

          if (coefmiddle==1)
            j=j-1;
          else
            j=j+1;              // read next coefficient

            b=b-1;
                  // after counter for read address b is 0, then b=74
            if (b==127)  b=74;
                  // activate the binary to MDLNS converter
            if (k==255)  activate=1;
            if (k==256) activate=0;
        end

      k=k+1;
    end
end
endmodule
```

coef152.v

2-digit MDLNS coefficient ROM

```
coefsignrom1[0]=2'b01;coefbinrom1[0]=6'b101111;coefternrom1[0]=2'b01;
coefsignrom1[1]=2'b01;coefbinrom1[1]=6'b110000;coefternrom1[1]=2'b00;
coefsignrom1[2]=2'b01;coefbinrom1[2]=6'b110101;coefternrom1[2]=2'b11;
coefsignrom1[3]=2'b01;coefbinrom1[3]=6'b110101;coefternrom1[3]=2'b00;
coefsignrom1[4]=2'b01;coefbinrom1[4]=6'b110110;coefternrom1[4]=2'b10;
coefsignrom1[5]=2'b01;coefbinrom1[5]=6'b110110;coefternrom1[5]=2'b00;
coefsignrom1[6]=2'b01;coefbinrom1[6]=6'b110110;coefternrom1[6]=2'b11;
coefsignrom1[7]=2'b01;coefbinrom1[7]=6'b110110;coefternrom1[7]=2'b10;
coefsignrom1[8]=2'b01;coefbinrom1[8]=6'b110101;coefternrom1[8]=2'b11;
coefsignrom1[9]=2'b00;coefbinrom1[9]=6'b000000;coefternrom1[9]=2'b00;
coefsignrom1[10]=2'b11;coefbinrom1[10]=6'b111000;coefternrom1[10]=2'b11;
coefsignrom1[11]=2'b11;coefbinrom1[11]=6'b110010;coefternrom1[11]=2'b00;
coefsignrom1[12]=2'b11;coefbinrom1[12]=6'b110111;coefternrom1[12]=2'b11;
coefsignrom1[13]=2'b11;coefbinrom1[13]=6'b111000;coefternrom1[13]=2'b11;
coefsignrom1[14]=2'b11;coefbinrom1[14]=6'b111001;coefternrom1[14]=2'b10;
coefsignrom1[15]=2'b11;coefbinrom1[15]=6'b110101;coefternrom1[15]=2'b00;
```

```
coefsignrom1[16]=2'b11;coefbinrom1[16]=6'b111000;coefternrom1[16]=2'b01;
coefsignrom1[17]=2'b11;coefbinrom1[17]=6'b111000;coefternrom1[17]=2'b11;
coefsignrom1[18]=2'b11;coefbinrom1[18]=6'b111001;coefternrom1[18]=2'b11;
coefsignrom1[19]=2'b11;coefbinrom1[19]=6'b111001;coefternrom1[19]=2'b11;
coefsignrom1[20]=2'b11;coefbinrom1[20]=6'b111001;coefternrom1[20]=2'b00;
coefsignrom1[21]=2'b11;coefbinrom1[21]=6'b110101;coefternrom1[21]=2'b00;
coefsignrom1[22]=2'b11;coefbinrom1[22]=6'b111000;coefternrom1[22]=2'b11;
coefsignrom1[23]=2'b00;coefbinrom1[23]=6'b000000;coefternrom1[23]=2'b00;
coefsignrom1[24]=2'b01;coefbinrom1[24]=6'b111001;coefternrom1[24]=2'b01;
coefsignrom1[25]=2'b01;coefbinrom1[25]=6'b111011;coefternrom1[25]=2'b11;
coefsignrom1[26]=2'b01;coefbinrom1[26]=6'b111011;coefternrom1[26]=2'b11;
coefsignrom1[27]=2'b01;coefbinrom1[27]=6'b111010;coefternrom1[27]=2'b01;
coefsignrom1[28]=2'b01;coefbinrom1[28]=6'b111011;coefternrom1[28]=2'b00;
coefsignrom1[29]=2'b01;coefbinrom1[29]=6'b111011;coefternrom1[29]=2'b11;
coefsignrom1[30]=2'b01;coefbinrom1[30]=6'b111011;coefternrom1[30]=2'b01;
coefsignrom1[31]=2'b01;coefbinrom1[31]=6'b111100;coefternrom1[31]=2'b11;
coefsignrom1[32]=2'b01;coefbinrom1[32]=6'b111100;coefternrom1[32]=2'b00;
coefsignrom1[33]=2'b01;coefbinrom1[33]=6'b111101;coefternrom1[33]=2'b10;
coefsignrom1[34]=2'b01;coefbinrom1[34]=6'b111100;coefternrom1[34]=2'b00;
coefsignrom1[35]=2'b01;coefbinrom1[35]=6'b111001;coefternrom1[35]=2'b11;
coefsignrom1[36]=2'b01;coefbinrom1[36]=6'b111100;coefternrom1[36]=2'b01;
coefsignrom1[37]=2'b01;coefbinrom1[37]=6'b111011;coefternrom1[37]=2'b00;
coefsignrom1[38]=2'b11;coefbinrom1[38]=6'b110100;coefternrom1[38]=2'b00;
coefsignrom1[39]=2'b11;coefbinrom1[39]=6'b110101;coefternrom1[39]=2'b00;
coefsignrom1[40]=2'b11;coefbinrom1[40]=6'b110110;coefternrom1[40]=2'b11;
coefsignrom1[41]=2'b11;coefbinrom1[41]=6'b110110;coefternrom1[41]=2'b00;
coefsignrom1[42]=2'b11;coefbinrom1[42]=6'b110110;coefternrom1[42]=2'b11;
coefsignrom1[43]=2'b11;coefbinrom1[43]=6'b110101;coefternrom1[43]=2'b11;
coefsignrom1[44]=2'b01;coefbinrom1[44]=6'b110101;coefternrom1[44]=2'b10;
coefsignrom1[45]=2'b01;coefbinrom1[45]=6'b110110;coefternrom1[45]=2'b11;
coefsignrom1[46]=2'b01;coefbinrom1[46]=6'b110110;coefternrom1[46]=2'b10;
coefsignrom1[47]=2'b00;coefbinrom1[47]=6'b000000;coefternrom1[47]=2'b00;
coefsignrom1[48]=2'b11;coefbinrom1[48]=6'b110110;coefternrom1[48]=2'b00;
coefsignrom1[49]=2'b11;coefbinrom1[49]=6'b110111;coefternrom1[49]=2'b11;
coefsignrom1[50]=2'b11;coefbinrom1[50]=6'b110111;coefternrom1[50]=2'b10;
coefsignrom1[51]=2'b01;coefbinrom1[51]=6'b110101;coefternrom1[51]=2'b01;
coefsignrom1[52]=2'b01;coefbinrom1[52]=6'b111000;coefternrom1[52]=2'b01;
coefsignrom1[53]=2'b01;coefbinrom1[53]=6'b111010;coefternrom1[53]=2'b10;
coefsignrom1[54]=2'b01;coefbinrom1[54]=6'b111001;coefternrom1[54]=2'b01;
coefsignrom1[55]=2'b01;coefbinrom1[55]=6'b111010;coefternrom1[55]=2'b11;
coefsignrom1[56]=2'b01;coefbinrom1[56]=6'b111000;coefternrom1[56]=2'b11;
coefsignrom1[57]=2'b01;coefbinrom1[57]=6'b110100;coefternrom1[57]=2'b11;
coefsignrom1[58]=2'b11;coefbinrom1[58]=6'b111000;coefternrom1[58]=2'b11;
coefsignrom1[59]=2'b11;coefbinrom1[59]=6'b111000;coefternrom1[59]=2'b01;
coefsignrom1[60]=2'b11;coefbinrom1[60]=6'b111001;coefternrom1[60]=2'b11;
coefsignrom1[61]=2'b00;coefbinrom1[61]=6'b000000;coefternrom1[61]=2'b00;
coefsignrom1[62]=2'b01;coefbinrom1[62]=6'b110110;coefternrom1[62]=2'b00;
coefsignrom1[63]=2'b01;coefbinrom1[63]=6'b111010;coefternrom1[63]=2'b00;
coefsignrom1[64]=2'b01;coefbinrom1[64]=6'b110101;coefternrom1[64]=2'b10;
coefsignrom1[65]=2'b11;coefbinrom1[65]=6'b111010;coefternrom1[65]=2'b10;
coefsignrom1[66]=2'b11;coefbinrom1[66]=6'b111100;coefternrom1[66]=2'b10;
coefsignrom1[67]=2'b11;coefbinrom1[67]=6'b111100;coefternrom1[67]=2'b00;
coefsignrom1[68]=2'b11;coefbinrom1[68]=6'b111100;coefternrom1[68]=2'b01;
coefsignrom1[69]=2'b11;coefbinrom1[69]=6'b111101;coefternrom1[69]=2'b00;
```

```
coefsignrom1[70]=2'b11;coefbinrom1[70]=6'b000000;coefternrom1[70]=2'b01;
coefsignrom1[71]=2'b11;coefbinrom1[71]=6'b111011;coefternrom1[71]=2'b11;
coefsignrom1[72]=2'b01;coefbinrom1[72]=6'b111001;coefternrom1[72]=2'b01;
coefsignrom1[73]=2'b01;coefbinrom1[73]=6'b111101;coefternrom1[73]=2'b10;
coefsignrom1[74]=2'b01;coefbinrom1[74]=6'b111101;coefternrom1[74]=2'b00;
coefsignrom1[75]=2'b01;coefbinrom1[75]=6'b111100;coefternrom1[75]=2'b00;
coefsignrom1[76]=2'b11;coefbinrom1[76]=6'b110011;coefternrom1[76]=2'b10;
coefsignrom1[77]=2'b01;coefbinrom1[77]=6'b110100;coefternrom1[77]=2'b01;
coefsignrom1[78]=2'b01;coefbinrom1[78]=6'b110110;coefternrom1[78]=2'b11;
coefsignrom1[79]=2'b01;coefbinrom1[79]=6'b110101;coefternrom1[79]=2'b01;
coefsignrom1[80]=2'b11;coefbinrom1[80]=6'b110011;coefternrom1[80]=2'b00;
coefsignrom1[81]=2'b11;coefbinrom1[81]=6'b110111;coefternrom1[81]=2'b10;
coefsignrom1[82]=2'b11;coefbinrom1[82]=6'b110110;coefternrom1[82]=2'b01;
coefsignrom1[83]=2'b11;coefbinrom1[83]=6'b110100;coefternrom1[83]=2'b00;
coefsignrom1[84]=2'b01;coefbinrom1[84]=6'b110100;coefternrom1[84]=2'b10;
coefsignrom1[85]=2'b00;coefbinrom1[85]=6'b000000;coefternrom1[85]=2'b00;
coefsignrom1[86]=2'b11;coefbinrom1[86]=6'b110010;coefternrom1[86]=2'b01;
coefsignrom1[87]=2'b01;coefbinrom1[87]=6'b110110;coefternrom1[87]=2'b10;
coefsignrom1[88]=2'b01;coefbinrom1[88]=6'b111000;coefternrom1[88]=2'b00;
coefsignrom1[89]=2'b01;coefbinrom1[89]=6'b111001;coefternrom1[89]=2'b11;
coefsignrom1[90]=2'b01;coefbinrom1[90]=6'b110111;coefternrom1[90]=2'b00;
coefsignrom1[91]=2'b11;coefbinrom1[91]=6'b111000;coefternrom1[91]=2'b01;
coefsignrom1[92]=2'b11;coefbinrom1[92]=6'b111001;coefternrom1[92]=2'b01;
coefsignrom1[93]=2'b11;coefbinrom1[93]=6'b111000;coefternrom1[93]=2'b00;
coefsignrom1[94]=2'b01;coefbinrom1[94]=6'b110111;coefternrom1[94]=2'b01;
coefsignrom1[95]=2'b01;coefbinrom1[95]=6'b111010;coefternrom1[95]=2'b00;
coefsignrom1[96]=2'b01;coefbinrom1[96]=6'b111010;coefternrom1[96]=2'b11;
coefsignrom1[97]=2'b01;coefbinrom1[97]=6'b111000;coefternrom1[97]=2'b10;
coefsignrom1[98]=2'b11;coefbinrom1[98]=6'b111000;coefternrom1[98]=2'b11;
coefsignrom1[99]=2'b00;coefbinrom1[99]=6'b000000;coefternrom1[99]=2'b00;
coefsignrom1[100]=2'b01;coefbinrom1[100]=6'b111001;
coefternrom1[100]=2'b11;
coefsignrom1[101]=2'b11;coefbinrom1[101]=6'b111000;
coefternrom1[101]=2'b00;
coefsignrom1[102]=2'b11;coefbinrom1[102]=6'b111100;
coefternrom1[102]=2'b10;
coefsignrom1[103]=2'b11;coefbinrom1[103]=6'b111011;
coefternrom1[103]=2'b01;
coefsignrom1[104]=2'b11;coefbinrom1[104]=6'b111010;
coefternrom1[104]=2'b11;
coefsignrom1[105]=2'b01;coefbinrom1[105]=6'b111000;
coefternrom1[105]=2'b01;
coefsignrom1[106]=2'b01;coefbinrom1[106]=6'b111100;
coefternrom1[106]=2'b01;
coefsignrom1[107]=2'b01;coefbinrom1[107]=6'b111101;
coefternrom1[107]=2'b10;
coefsignrom1[108]=2'b11;coefbinrom1[108]=6'b111011;
coefternrom1[108]=2'b11;
coefsignrom1[109]=2'b11;coefbinrom1[109]=6'b111111;
coefternrom1[109]=2'b10;
coefsignrom1[110]=2'b11;coefbinrom1[110]=6'b111101;
coefternrom1[110]=2'b00;
coefsignrom1[111]=2'b11;coefbinrom1[111]=6'b111011;
coefternrom1[111]=2'b00;
```

```
coefsignrom1[112]=2'b01;coefbinrom1[112]=6'b111101;
coefternrom1[112]=2'b10;
coefsignrom1[113]=2'b01;coefbinrom1[113]=6'b111100;
coefternrom1[113]=2'b00;
coefsignrom1[114]=2'b01;coefbinrom1[114]=6'b110101;
coefternrom1[114]=2'b10;
coefsignrom1[115]=2'b11;coefbinrom1[115]=6'b110010;
coefternrom1[115]=2'b01;
coefsignrom1[116]=2'b11;coefbinrom1[116]=6'b110110;
coefternrom1[116]=2'b11;
coefsignrom1[117]=2'b11;coefbinrom1[117]=6'b110100;
coefternrom1[117]=2'b10;
coefsignrom1[118]=2'b01;coefbinrom1[118]=6'b110110;
coefternrom1[118]=2'b01;
coefsignrom1[119]=2'b01;coefbinrom1[119]=6'b111000;
coefternrom1[119]=2'b10;
coefsignrom1[120]=2'b11;coefbinrom1[120]=6'b111000;
coefternrom1[120]=2'b10;
coefsignrom1[121]=2'b11;coefbinrom1[121]=6'b110111;
coefternrom1[121]=2'b10;
coefsignrom1[122]=2'b01;coefbinrom1[122]=6'b110100;
coefternrom1[122]=2'b01;
coefsignrom1[123]=2'b00;coefbinrom1[123]=6'b000000;
coefternrom1[123]=2'b00;
coefsignrom1[124]=2'b11;coefbinrom1[124]=6'b110100;
coefternrom1[124]=2'b00;
coefsignrom1[125]=2'b01;coefbinrom1[125]=6'b110111;
coefternrom1[125]=2'b00;
coefsignrom1[126]=2'b01;coefbinrom1[126]=6'b111000;
coefternrom1[126]=2'b01;
coefsignrom1[127]=2'b11;coefbinrom1[127]=6'b111000;
coefternrom1[127]=2'b00;
coefsignrom1[128]=2'b11;coefbinrom1[128]=6'b111000;
coefternrom1[128]=2'b01;
coefsignrom1[129]=2'b01;coefbinrom1[129]=6'b111000;
coefternrom1[129]=2'b10;
coefsignrom1[130]=2'b01;coefbinrom1[130]=6'b111001;
coefternrom1[130]=2'b01;
coefsignrom1[131]=2'b01;coefbinrom1[131]=6'b111000;
coefternrom1[131]=2'b11;
coefsignrom1[132]=2'b11;coefbinrom1[132]=6'b111010;
coefternrom1[132]=2'b00;
coefsignrom1[133]=2'b11;coefbinrom1[133]=6'b111001;
coefternrom1[133]=2'b00;
coefsignrom1[134]=2'b01;coefbinrom1[134]=6'b111010;
coefternrom1[134]=2'b10;
coefsignrom1[135]=2'b01;coefbinrom1[135]=6'b111010;
coefternrom1[135]=2'b10;
coefsignrom1[136]=2'b11;coefbinrom1[136]=6'b110110;
coefternrom1[136]=2'b00;
coefsignrom1[137]=2'b00;coefbinrom1[137]=6'b000000;
coefternrom1[137]=2'b00;
coefsignrom1[138]=2'b01;coefbinrom1[138]=6'b111000;
coefternrom1[138]=2'b10;
```

```
coefsignrom1[139]=2'b11;coefbinrom1[139]=6'b111011;
coefternrom1[139]=2'b01;
coefsignrom1[140]=2'b11;coefbinrom1[140]=6'b111001;
coefternrom1[140]=2'b00;
coefsignrom1[141]=2'b01;coefbinrom1[141]=6'b111001;
coefternrom1[141]=2'b01;
coefsignrom1[142]=2'b01;coefbinrom1[142]=6'b111100;
coefternrom1[142]=2'b11;
coefsignrom1[143]=2'b11;coefbinrom1[143]=6'b111010;
coefternrom1[143]=2'b01;
coefsignrom1[144]=2'b11;coefbinrom1[144]=6'b111100;
coefternrom1[144]=2'b01;
coefsignrom1[145]=2'b11;coefbinrom1[145]=6'b111101;
coefternrom1[145]=2'b11;
coefsignrom1[146]=2'b01;coefbinrom1[146]=6'b111101;
coefternrom1[146]=2'b11;
coefsignrom1[147]=2'b01;coefbinrom1[147]=6'b111101;
coefternrom1[147]=2'b10;
coefsignrom1[148]=2'b11;coefbinrom1[148]=6'b111100;
coefternrom1[148]=2'b01;
coefsignrom1[149]=2'b11;coefbinrom1[149]=6'b111101;
coefternrom1[149]=2'b00;
coefsignrom1[150]=2'b01;coefbinrom1[150]=6'b111100;
coefternrom1[150]=2'b01;
coefsignrom1[151]=2'b01;coefbinrom1[151]=6'b111100;
coefternrom1[151]=2'b00;
coefsignrom2[0]=2'b01;coefbinrom2[0]=6'b110100;coefternrom2[0]=2'b10;
coefsignrom2[1]=2'b01;coefbinrom2[1]=6'b110100;coefternrom2[1]=2'b00;
coefsignrom2[2]=2'b11;coefbinrom2[2]=6'b101110;coefternrom2[2]=2'b11;
coefsignrom2[3]=2'b01;coefbinrom2[3]=6'b110000;coefternrom2[3]=2'b11;
coefsignrom2[4]=2'b01;coefbinrom2[4]=6'b110000;coefternrom2[4]=2'b00;
coefsignrom2[5]=2'b11;coefbinrom2[5]=6'b110101;coefternrom2[5]=2'b10;
coefsignrom2[6]=2'b11;coefbinrom2[6]=6'b110010;coefternrom2[6]=2'b01;
coefsignrom2[7]=2'b11;coefbinrom2[7]=6'b101110;coefternrom2[7]=2'b11;
coefsignrom2[8]=2'b11;coefbinrom2[8]=6'b101111;coefternrom2[8]=2'b01;
coefsignrom2[9]=2'b00;coefbinrom2[9]=6'b000000;coefternrom2[9]=2'b00;
coefsignrom2[10]=2'b01;coefbinrom2[10]=6'b110111;coefternrom2[10]=2'b01;
coefsignrom2[11]=2'b11;coefbinrom2[11]=6'b110111;coefternrom2[11]=2'b10;
coefsignrom2[12]=2'b11;coefbinrom2[12]=6'b110110;coefternrom2[12]=2'b10;
coefsignrom2[13]=2'b11;coefbinrom2[13]=6'b110010;coefternrom2[13]=2'b00;
coefsignrom2[14]=2'b01;coefbinrom2[14]=6'b110110;coefternrom2[14]=2'b10;
coefsignrom2[15]=2'b11;coefbinrom2[15]=6'b111001;coefternrom2[15]=2'b10;
coefsignrom2[16]=2'b11;coefbinrom2[16]=6'b110111;coefternrom2[16]=2'b10;
coefsignrom2[17]=2'b11;coefbinrom2[17]=6'b111000;coefternrom2[17]=2'b00;
coefsignrom2[18]=2'b11;coefbinrom2[18]=6'b110110;coefternrom2[18]=2'b01;
coefsignrom2[19]=2'b11;coefbinrom2[19]=6'b110111;coefternrom2[19]=2'b10;
coefsignrom2[20]=2'b01;coefbinrom2[20]=6'b110111;coefternrom2[20]=2'b10;
coefsignrom2[21]=2'b11;coefbinrom2[21]=6'b111001;coefternrom2[21]=2'b10;
coefsignrom2[22]=2'b11;coefbinrom2[22]=6'b101110;coefternrom2[22]=2'b11;
coefsignrom2[23]=2'b00;coefbinrom2[23]=6'b000000;coefternrom2[23]=2'b00;
coefsignrom2[24]=2'b11;coefbinrom2[24]=6'b111001;coefternrom2[24]=2'b11;
coefsignrom2[25]=2'b11;coefbinrom2[25]=6'b111010;coefternrom2[25]=2'b00;
coefsignrom2[26]=2'b11;coefbinrom2[26]=6'b111001;coefternrom2[26]=2'b01;
coefsignrom2[27]=2'b01;coefbinrom2[27]=6'b110101;coefternrom2[27]=2'b00;
```

```
coefsignrom2[28]=2'b11;coefbinrom2[28]=6'b111000;coefternrom2[28]=2'b00;
coefsignrom2[29]=2'b01;coefbinrom2[29]=6'b111001;coefternrom2[29]=2'b01;
coefsignrom2[30]=2'b01;coefbinrom2[30]=6'b110110;coefternrom2[30]=2'b01;
coefsignrom2[31]=2'b11;coefbinrom2[31]=6'b110101;coefternrom2[31]=2'b11;
coefsignrom2[32]=2'b11;coefbinrom2[32]=6'b111001;coefternrom2[32]=2'b00;
coefsignrom2[33]=2'b11;coefbinrom2[33]=6'b111010;coefternrom2[33]=2'b00;
coefsignrom2[34]=2'b01;coefbinrom2[34]=6'b110111;coefternrom2[34]=2'b01;
coefsignrom2[35]=2'b01;coefbinrom2[35]=6'b111100;coefternrom2[35]=2'b00;
coefsignrom2[36]=2'b11;coefbinrom2[36]=6'b111010;coefternrom2[36]=2'b10;
coefsignrom2[37]=2'b01;coefbinrom2[37]=6'b111011;coefternrom2[37]=2'b01;
coefsignrom2[38]=2'b01;coefbinrom2[38]=6'b110010;coefternrom2[38]=2'b11;
coefsignrom2[39]=2'b01;coefbinrom2[39]=6'b110000;coefternrom2[39]=2'b01;
coefsignrom2[40]=2'b01;coefbinrom2[40]=6'b101111;coefternrom2[40]=2'b11;
coefsignrom2[41]=2'b01;coefbinrom2[41]=6'b110011;coefternrom2[41]=2'b10;
coefsignrom2[42]=2'b01;coefbinrom2[42]=6'b101101;coefternrom2[42]=2'b00;
coefsignrom2[43]=2'b01;coefbinrom2[43]=6'b110010;coefternrom2[43]=2'b01;
coefsignrom2[44]=2'b01;coefbinrom2[44]=6'b101111;coefternrom2[44]=2'b00;
coefsignrom2[45]=2'b11;coefbinrom2[45]=6'b110001;coefternrom2[45]=2'b11;
coefsignrom2[46]=2'b01;coefbinrom2[46]=6'b110011;coefternrom2[46]=2'b10;
coefsignrom2[47]=2'b00;coefbinrom2[47]=6'b000000;coefternrom2[47]=2'b00;
coefsignrom2[48]=2'b01;coefbinrom2[48]=6'b101111;coefternrom2[48]=2'b11;
coefsignrom2[49]=2'b11;coefbinrom2[49]=6'b110001;coefternrom2[49]=2'b00;
coefsignrom2[50]=2'b01;coefbinrom2[50]=6'b110100;coefternrom2[50]=2'b00;
coefsignrom2[51]=2'b01;coefbinrom2[51]=6'b110110;coefternrom2[51]=2'b11;
coefsignrom2[52]=2'b01;coefbinrom2[52]=6'b110100;coefternrom2[52]=2'b11;
coefsignrom2[53]=2'b11;coefbinrom2[53]=6'b110010;coefternrom2[53]=2'b00;
coefsignrom2[54]=2'b01;coefbinrom2[54]=6'b111000;coefternrom2[54]=2'b10;
coefsignrom2[55]=2'b01;coefbinrom2[55]=6'b110100;coefternrom2[55]=2'b01;
coefsignrom2[56]=2'b01;coefbinrom2[56]=6'b111001;coefternrom2[56]=2'b11;
coefsignrom2[57]=2'b01;coefbinrom2[57]=6'b111000;coefternrom2[57]=2'b11;
coefsignrom2[58]=2'b01;coefbinrom2[58]=6'b110011;coefternrom2[58]=2'b11;
coefsignrom2[59]=2'b11;coefbinrom2[59]=6'b110111;coefternrom2[59]=2'b11;
coefsignrom2[60]=2'b01;coefbinrom2[60]=6'b110110;coefternrom2[60]=2'b10;
coefsignrom2[61]=2'b00;coefbinrom2[61]=6'b000000;coefternrom2[61]=2'b00;
coefsignrom2[62]=2'b01;coefbinrom2[62]=6'b111001;coefternrom2[62]=2'b11;
coefsignrom2[63]=2'b11;coefbinrom2[63]=6'b111001;coefternrom2[63]=2'b10;
coefsignrom2[64]=2'b01;coefbinrom2[64]=6'b111001;coefternrom2[64]=2'b11;
coefsignrom2[65]=2'b01;coefbinrom2[65]=6'b110100;coefternrom2[65]=2'b01;
coefsignrom2[66]=2'b01;coefbinrom2[66]=6'b111000;coefternrom2[66]=2'b00;
coefsignrom2[67]=2'b01;coefbinrom2[67]=6'b110110;coefternrom2[67]=2'b10;
coefsignrom2[68]=2'b11;coefbinrom2[68]=6'b110111;coefternrom2[68]=2'b01;
coefsignrom2[69]=2'b01;coefbinrom2[69]=6'b111100;coefternrom2[69]=2'b10;
coefsignrom2[70]=2'b01;coefbinrom2[70]=6'b000001;coefternrom2[70]=2'b10;
coefsignrom2[71]=2'b11;coefbinrom2[71]=6'b110111;coefternrom2[71]=2'b01;
coefsignrom2[72]=2'b01;coefbinrom2[72]=6'b111011;coefternrom2[72]=2'b10;
coefsignrom2[73]=2'b01;coefbinrom2[73]=6'b111010;coefternrom2[73]=2'b10;
coefsignrom2[74]=2'b01;coefbinrom2[74]=6'b111000;coefternrom2[74]=2'b10;
coefsignrom2[75]=2'b01;coefbinrom2[75]=6'b111100;coefternrom2[75]=2'b01;
coefsignrom2[76]=2'b01;coefbinrom2[76]=6'b101110;coefternrom2[76]=2'b00;
coefsignrom2[77]=2'b01;coefbinrom2[77]=6'b101111;coefternrom2[77]=2'b01;
coefsignrom2[78]=2'b11;coefbinrom2[78]=6'b101111;coefternrom2[78]=2'b11;
coefsignrom2[79]=2'b11;coefbinrom2[79]=6'b101110;coefternrom2[79]=2'b11;
coefsignrom2[80]=2'b11;coefbinrom2[80]=6'b110100;coefternrom2[80]=2'b10;
coefsignrom2[81]=2'b11;coefbinrom2[81]=6'b110010;coefternrom2[81]=2'b10;
```

```
coefsignrom2[82]=2'b01;coefbinrom2[82]=6'b110001;coefternrom2[82]=2'b11;
coefsignrom2[83]=2'b11;coefbinrom2[83]=6'b110001;coefternrom2[83]=2'b10;
coefsignrom2[84]=2'b01;coefbinrom2[84]=6'b110100;coefternrom2[84]=2'b01;
coefsignrom2[85]=2'b00;coefbinrom2[85]=6'b000000;coefternrom2[85]=2'b00;
coefsignrom2[86]=2'b11;coefbinrom2[86]=6'b110110;coefternrom2[86]=2'b10;
coefsignrom2[87]=2'b11;coefbinrom2[87]=6'b110010;coefternrom2[87]=2'b10;
coefsignrom2[88]=2'b11;coefbinrom2[88]=6'b110011;coefternrom2[88]=2'b11;
coefsignrom2[89]=2'b11;coefbinrom2[89]=6'b110101;coefternrom2[89]=2'b00;
coefsignrom2[90]=2'b11;coefbinrom2[90]=6'b110011;coefternrom2[90]=2'b11;
coefsignrom2[91]=2'b11;coefbinrom2[91]=6'b110111;coefternrom2[91]=2'b11;
coefsignrom2[92]=2'b11;coefbinrom2[92]=6'b111000;coefternrom2[92]=2'b10;
coefsignrom2[93]=2'b11;coefbinrom2[93]=6'b111001;coefternrom2[93]=2'b10;
coefsignrom2[94]=2'b01;coefbinrom2[94]=6'b110110;coefternrom2[94]=2'b11;
coefsignrom2[95]=2'b11;coefbinrom2[95]=6'b110111;coefternrom2[95]=2'b01;
coefsignrom2[96]=2'b11;coefbinrom2[96]=6'b110101;coefternrom2[96]=2'b10;
coefsignrom2[97]=2'b11;coefbinrom2[97]=6'b110010;coefternrom2[97]=2'b11;
coefsignrom2[98]=2'b11;coefbinrom2[98]=6'b110110;coefternrom2[98]=2'b11;
coefsignrom2[99]=2'b00;coefbinrom2[99]=6'b000000;coefternrom2[99]=2'b00;
coefsignrom2[100]=2'b11;coefbinrom2[100]=6'b110111;
coefternrom2[100]=2'b10;
coefsignrom2[101]=2'b01;coefbinrom2[101]=6'b110000;
coefternrom2[101]=2'b01;
coefsignrom2[102]=2'b01;coefbinrom2[102]=6'b111010;
coefternrom2[102]=2'b11;
coefsignrom2[103]=2'b01;coefbinrom2[103]=6'b111000;
coefternrom2[103]=2'b11;
coefsignrom2[104]=2'b01;coefbinrom2[104]=6'b110000;
coefternrom2[104]=2'b00;
coefsignrom2[105]=2'b01;coefbinrom2[105]=6'b111100;
coefternrom2[105]=2'b10;
coefsignrom2[106]=2'b01;coefbinrom2[106]=6'b110111;
coefternrom2[106]=2'b01;
coefsignrom2[107]=2'b11;coefbinrom2[107]=6'b111010;
coefternrom2[107]=2'b00;
coefsignrom2[108]=2'b01;coefbinrom2[108]=6'b101011;
coefternrom2[108]=2'b01;
coefsignrom2[109]=2'b01;coefbinrom2[109]=6'b111110;
coefternrom2[109]=2'b11;
coefsignrom2[110]=2'b01;coefbinrom2[110]=6'b111001;
coefternrom2[110]=2'b00;
coefsignrom2[111]=2'b01;coefbinrom2[111]=6'b110110;
coefternrom2[111]=2'b11;
coefsignrom2[112]=2'b01;coefbinrom2[112]=6'b111010;
coefternrom2[112]=2'b11;
coefsignrom2[113]=2'b01;coefbinrom2[113]=6'b111100;
coefternrom2[113]=2'b01;
coefsignrom2[114]=2'b11;coefbinrom2[114]=6'b110000;
coefternrom2[114]=2'b11;
coefsignrom2[115]=2'b11;coefbinrom2[115]=6'b110010;
coefternrom2[115]=2'b10;
coefsignrom2[116]=2'b01;coefbinrom2[116]=6'b101111;
coefternrom2[116]=2'b11;
coefsignrom2[117]=2'b11;coefbinrom2[117]=6'b110011;
coefternrom2[117]=2'b10;
```

```
coefsignrom2[118]=2'b11;coefbinrom2[118]=6'b110011;
coefternrom2[118]=2'b01;
coefsignrom2[119]=2'b11;coefbinrom2[119]=6'b110111;
coefternrom2[119]=2'b11;
coefsignrom2[120]=2'b01;coefbinrom2[120]=6'b110111;
coefternrom2[120]=2'b11;
coefsignrom2[121]=2'b01;coefbinrom2[121]=6'b110011;
coefternrom2[121]=2'b00;
coefsignrom2[122]=2'b11;coefbinrom2[122]=6'b110100;
coefternrom2[122]=2'b10;
coefsignrom2[123]=2'b00;coefbinrom2[123]=6'b000000;
coefternrom2[123]=2'b00;
coefsignrom2[124]=2'b01;coefbinrom2[124]=6'b101111;
coefternrom2[124]=2'b10;
coefsignrom2[125]=2'b01;coefbinrom2[125]=6'b110101;
coefternrom2[125]=2'b10;
coefsignrom2[126]=2'b11;coefbinrom2[126]=6'b111000;
coefternrom2[126]=2'b10;
coefsignrom2[127]=2'b01;coefbinrom2[127]=6'b110001;
coefternrom2[127]=2'b00;
coefsignrom2[128]=2'b11;coefbinrom2[128]=6'b110111;
coefternrom2[128]=2'b01;
coefsignrom2[129]=2'b11;coefbinrom2[129]=6'b110010;
coefternrom2[129]=2'b11;
coefsignrom2[130]=2'b01;coefbinrom2[130]=6'b111000;
coefternrom2[130]=2'b10;
coefsignrom2[131]=2'b01;coefbinrom2[131]=6'b110001;
coefternrom2[131]=2'b01;
coefsignrom2[132]=2'b01;coefbinrom2[132]=6'b111000;
coefternrom2[132]=2'b10;
coefsignrom2[133]=2'b11;coefbinrom2[133]=6'b110110;
coefternrom2[133]=2'b01;
coefsignrom2[134]=2'b11;coefbinrom2[134]=6'b110110;
coefternrom2[134]=2'b01;
coefsignrom2[135]=2'b11;coefbinrom2[135]=6'b110010;
coefternrom2[135]=2'b00;
coefsignrom2[136]=2'b11;coefbinrom2[136]=6'b110101;
coefternrom2[136]=2'b11;
coefsignrom2[137]=2'b00;coefbinrom2[137]=6'b000000;
coefternrom2[137]=2'b00;
coefsignrom2[138]=2'b11;coefbinrom2[138]=6'b110101;
coefternrom2[138]=2'b01;
coefsignrom2[139]=2'b01;coefbinrom2[139]=6'b111011;
coefternrom2[139]=2'b11;
coefsignrom2[140]=2'b11;coefbinrom2[140]=6'b111001;
coefternrom2[140]=2'b01;
coefsignrom2[141]=2'b01;coefbinrom2[141]=6'b111010;
coefternrom2[141]=2'b00;
coefsignrom2[142]=2'b01;coefbinrom2[142]=6'b110101;
coefternrom2[142]=2'b00;
coefsignrom2[143]=2'b01;coefbinrom2[143]=6'b111001;
coefternrom2[143]=2'b10;
coefsignrom2[144]=2'b11;coefbinrom2[144]=6'b110111;
coefternrom2[144]=2'b01;
```

```
coefsignrom2[145]=2'b01;coefbinrom2[145]=6'b111101;
coefternrom2[145]=2'b10;
coefsignrom2[146]=2'b01;coefbinrom2[146]=6'b110111;
coefternrom2[146]=2'b10;
coefsignrom2[147]=2'b11;coefbinrom2[147]=6'b110101;
coefternrom2[147]=2'b01;
coefsignrom2[148]=2'b11;coefbinrom2[148]=6'b110110;
coefternrom2[148]=2'b00;
coefsignrom2[149]=2'b01;coefbinrom2[149]=6'b110111;
coefternrom2[149]=2'b11;
coefsignrom2[150]=2'b11;coefbinrom2[150]=6'b111100;
coefternrom2[150]=2'b11;
coefsignrom2[151]=2'b01;coefbinrom2[151]=6'b111100;
coefternrom2[151]=2'b01;
```

B.3.3 p2s.v

```
module p2s(pinput1,pinput2,sout1,sout2,CK,reset,en,ready,start);

parameter aosize=18;

input [aosize-2:0] pinput1,pinput2;
input CK,en,reset;
output sout1,sout2,ready,start;

reg [4:0] out_count;
reg [aosize-2:0] par1,par2;
reg sout1,sout2, ready, start;

always @(posedge CK)
begin
    if (!reset)
        begin
            sout1 = 1'b0;
            sout2=0;
            par1=17'b0;
            par2=17'b0;
            ready=1'b0;
            out_count=0;
            start = 0;
        end
     else if(en)
        begin
            out_count=5'd0;
            start = 1;
        end
    else if(start==1 && out_count<5'd17)
        begin
            par1=pinput1;
            par2=pinput2;
            ready=1'b1;
            sout1 = par1[out_count];
            sout2 =par2[out_count];
            out_count=out_count+1;
        end
    else
        begin
            start = 0;
            ready=1'b0;
        end
end

endmodule
```

B.3.4 dbnsfilterchip_ram.v

```
//datapath of the MDLNS filterbank

module dbnsfilterchip_ram(out1,out2, CK,coefsign1, coefsign2,coefbin1,
coefbin2, coeftern1, coeftern2,  datasign1, datasign2,databin1, databin2
, datatern1, datatern2,clear1,reset,wout,j0);

parameter aosize=14,dbsize=4,dtsize=3,cbsize=4,ctsize=2,mbsize=5;
parameter mtsize=4, lbsize=10,ltsize=9,absize=10,sbsize=14,aisize=18;

input                   CK,clear1,reset;
input                   wout;
input  [1:0]            datasign1,datasign2,coefsign1,coefsign2;
input  [dbsize:0]       databin1,databin2;
input  [cbsize:0]       coefbin1,coefbin2;
input  [dtsize:0]       datatern1,datatern2;
input  [ctsize:0]       coeftern1,coeftern2;
input   j0;
output  [aosize-2:0]    out1,out2;

wire                    wout2,clear2;
wire [aosize:0]         regout1,regout2;
wire [aosize:0]         net112;
wire [sbsize+3:0]       sumout,net1;
wire  [aosize:0]        net045,net2,net3;

fourmac #(aisize,sbsize,aosize,dbsize,dtsize,cbsize,
          ctsize,mbsize,mtsize,lbsize,ltsize,absize)
fourmac1(sumout[sbsize+3:0], datasign1[1:0],databin1[dbsize:0],
         datatern1[dtsize:0], coefsign1[1:0], coefbin1[cbsize:0],
         coeftern1[ctsize:0], datasign2[1:0], databin2[dbsize:0],
         datatern2[dtsize:0], coefsign2[1:0], coefbin2[cbsize:0],
         coeftern2[ctsize:0],reset);
                        //Accumulator for channel 1
binaryadde#(aosize,sbsize+3,aosize)ACCUM(net112[aosize:0],sumout,net045);

rlatch_ram #(aosize) register ( CK,clear1, net045[aosize:0],
                                net112[aosize:0]);
rlatch_wout_ram #(aosize) r0  ( CK, reset, wout, net112[aosize:0],
                                regout1[aosize:0]);

                         //sign change register for channel 2
```

```
rlatch_channel2 #(sbsize+3) r1 (CK,reset,j0,sumout,net1);
                         //accumulator for channel 2
binaryadder #(sbsize+3,aosize,aosize) ACCUM2(net1,net3,net2);

rlatch_ram #(aosize) register2 ( CK,clear1, net2[aosize:0],
                                  net3[aosize:0]);

rlatch_wout_ram #(aosize) r3 ( CK, reset, wout, net3[aosize:0],
                                regout2[aosize:0]);

 assign out1=regout1[aosize:2];
 assign out2=regout2[aosize:2];

endmodule

//fourmac is the two digits(four channel) multiply and binaryout and
accumulate with the register

module fourmac(sumout,datasign1,databin1,datatern1,coefsign1,coefbin1,
coeftern1,datasign2,databin2,datatern2,coefsign2,coefbin2,coeftern2,res
et);

//Define parameter
parameter
aisize=30,sbsize=30,aosize=30,dbsize=4,dtsize=3,cbsize=4,ctsize=2,mbsiz
e=5,mtsize=4,lbsize=10,ltsize=9,absize=10;
//Define ports

input [1:0] datasign1 ,datasign2,coefsign1,coefsign2;
input [dbsize:0] databin1,databin2;
input [dtsize:0] datatern1,datatern2;
input [cbsize:0] coefbin1,coefbin2;
input [ctsize:0] coeftern1,coeftern2;
input reset;
output [sbsize+3:0] sumout;

//Define interconnections
wire [sbsize+1:0] ama1out,ama2out,ama3out,ama4out;
wire [sbsize+2:0] sum12,sum34;

//Instantiate ama
ama
#(ctsize,cbsize,dbsize,dtsize,mbsize,mtsize,lbsize,ltsize,absize,sbsize
)
ama1(ama1out,coefsign1,coefbin1,coeftern1,datasign1,databin1,datatern1,
reset);
ama
#(ctsize,cbsize,dbsize,dtsize,mbsize,mtsize,lbsize,ltsize,absize,sbsize
)
ama2(ama2out,coefsign1,coefbin1,coeftern1,datasign2,databin2,datatern2,
reset);
ama
#(ctsize,cbsize,dbsize,dtsize,mbsize,mtsize,lbsize,ltsize,absize,sbsize
)
```

```
ama3(ama3out,coefsign2,coefbin2,coeftern2,datasign1,databin1,datatern1,
reset);
ama
#(ctsize,cbsize,dbsize,dtsize,mbsize,mtsize,lbsize,ltsize,absize,sbsize
)
ama4(ama4out,coefsign2,coefbin2,coeftern2,datasign2,databin2,datatern2,
reset);

//Add them
binaryadder #(sbsize+1,sbsize+1,sbsize+2)
binaryadder12(ama1out,ama2out,sum12);
binaryadder #(sbsize+1,sbsize+1,sbsize+2)
binaryadder34(ama3out,ama4out,sum34);
binaryadder #(sbsize+2,sbsize+2,sbsize+3) binaryadder-
sum(sum12,sum34,sumout);

endmodule

// rlatch_channl2 is the sign change register for second output channel.
module rlatch_channel2(clock,clear,j0,indata,outdata);

                  //default parameter
parameter wsize=30;
                  //Defind ports
input  clock,clear;
input  j0;
input  [wsize:0] indata;
output [wsize:0] outdata;
reg [wsize:0] outdata;

                  //On rising adge of the clock ,passing the data
always @(posedge clock)
begin
    if(!clear)
       outdata = 0;
     //if the address of coefficient ROM is even number, change the sign
    else if (j0==0)     outdata = -indata;
    else outdata=indata;
end

endmodule

module rlatch_ram(clock,clear,indata,outdata);
                  //default parameter
parameter wsize=30;
                  //Defind ports
input  clock,clear;
input  [wsize:0] indata;
output [wsize:0] outdata;
reg [wsize:0] outdata;

                  //On rising adge of the clock ,passing the data
always @(posedge clock)
 begin
```

```
    if(!clear)
       outdata = 0;
    else  outdata = indata;
 end

endmodule

//ralch_wout_ram is a register only pass the data when output enabled
module rlatch_wout_ram(clock,clear,enable,indata,outdata);

                  //default parameter
parameter wsize=30;
                  //Defind ports
input  clock,clear;
input  enable;
input  [wsize:0] indata;
output [wsize:0] outdata;
reg [wsize:0] outdata;

                  //On rising adge of the clock ,passing the data
always @(posedge clock)
 begin
    if(!clear)
      outdata = 0;
    else if (enable)
      outdata = indata;
 end

endmodule
```

B.3.5 ama.v

ama.v is a module of MDLNS Mutiply and convert to binary out, which is the MDLNS MAC Cell, the most fundamental cell of the MDLNS filterbank.

```
module ama(signcorrout, coefsign, coefbin, coeftern, datasign, databin,
datatern,reset);

      //Setup default parameters
parameter  ctsize = 1 , cbsize=5 ,dbsize=5, dtsize=4, mbsize=6,mtsize=4;
parameter  lbsize=5, ltsize=16, absize=7,sbsize=15;
      //Define ports
output      [sbsize+1:0] signcorrout;
input       [1:0] coefsign;
input       [ctsize:0] coeftern;
input       [cbsize:0] coefbin;
input       [1:0] datasign;
input       [dbsize:0] databin;
input       [dtsize:0] datatern;
input       reset;
```

```
      //Define interconnections
wire [mbsize:0] resultbin;
wire [mtsize:0] resulttern;
wire [ltsize:0] lutvalue;
wire [lbsize:0] shiftvalue;
wire [absize:0] totalshift;
wire [sbsize:0] nosignvalue;

      // Instantiate multiplier
multiplier #(cbsize,ctsize,dbsize,dtsize,mbsize,mtsize) MUL(coef-
bin,coeftern,databin,datatern,resultbin,resulttern);
      //instantiate ternary look up table
ternarylut #(mtsize,ltsize,lbsize) TERNLUT(result-
tern,lutvalue,shiftvalue,reset);
      //Instantiate binary adder
binaryadder #(mbsize,lbsize,absize) BINADD(resultbin,shiftvalue,total-
shift);
      //instantiate barrel shift
bshifter #(ltsize,absize,sbsize) BSHIFT(lutvalue,totalshift,nosign-
value);
      //instantiate sign correction
signcorrect #(sbsize) CORRECT(coefsign,datasign,nosignvalue,signcor-
rout);

endmodule
```

multiplier.v

The MDLNS multiplier consists of two binary adders that add the ternary and binary exponents of the data and the coefficients

```
module multiplier (coefbin, coeftern, databin, datatern, resultbin,
resulttern);
      // Default parametersparameter cbsize = 7;
parameter ctsize = 7;
parameter dbsize = 7;
parameter dtsize = 7;
parameter mbsize = 8;
parameter mtsize = 8;

      // Define ports
input [cbsize:0] coefbin;
input [ctsize:0] coeftern;
input [dbsize:0] databin;
input [dtsize:0] datatern;
output [mbsize:0] resultbin;
output [mtsize:0] resulttern;
```

```
binaryadder #(cbsize,dbsize,mbsize) multi1(coefbin,databin,resultbin);
binaryadder #(ctsize,dtsize,mtsize) multi2(coeftern,datatern,result-
tern);

endmodule
```

binaryadder.v

binaryadder is a module for managing the binary exponents

```
module binaryadder(binary1,binary2,binaryout);

      //Default parameters
parameter mbsize=6;
parameter lbsize=5;
parameter absize=7;

      //Define ports
input [mbsize:0] binary1;
input [lbsize:0] binary2;
output [absize:0] binaryout;

wire[absize:0] arg1,arg2;

      //Extend signs on each argument
  assign arg1={{(absize-mbsize){binary1[mbsize]}},binary1[mbsize:0]};
  assign arg2={{(absize-lbsize){binary2[lbsize]}},binary2[lbsize:0]};
      // Add them
  assign binaryout=arg1+arg2;

endmodule
```

ternarylut.v

Module for obtaining a binary estimation of a MDLNS value from a lookup table (ROM in chapter 2). The lookup table is from Roberto Muscedere.

```
module ternarylut(ternaryin, ternaryout, binaryout,reset);

      //Default parameters
parameter mtsize=4;
parameter ltsize=9;
parameter lbsize=10;

      //Define ports
```

```
input [mtsize:0] ternaryin;
input reset;
output [ltsize:0] ternaryout;
output [lbsize:0] binaryout;

      //Setup array and intermediate register and read in look-up table
reg [ltsize+lbsize+1:0] ternrom[0:(1<<(mtsize+1))-1];

always @(reset)
  begin
      ternrom[5'b00000]=23'b10000000000000000110000;
      ternrom[5'b00001]=23'b10100100001111000110000;
      ternrom[5'b00010]=23'b11010010101110100110000;
      ternrom[5'b00011]=23'b10000111001100001110001;
      ternrom[5'b00100]=23'b10101101011101011110001;
      ternrom[5'b00101]=23'b11011110100100000110001;
      ternrom[5'b00110]=23'b10001110110010001110010;
      ternrom[5'b00111]=23'b10110111001101000110010;
      ternrom[5'b01000]=23'b11101011000100001110010;
      ternrom[5'b01001]=23'b10010110110011011110011;
      ternrom[5'b01010]=23'b11000001011111101110011;
      ternrom[5'b01011]=23'b11111000010001010110011;
      ternrom[5'b01100]=23'b10011111010001101110100;
      ternrom[5'b01101]=23'b11001100010111010110100;
      ternrom[5'b01110]=23'b10000011000110111110101;
      ternrom[5'b01111]=23'b10101000001110001110101;
      ternrom[5'b10000]=23'b10010111110100000101010;
      ternrom[5'b10001]=23'b11000010110010100101010;
      ternrom[5'b10010]=23'b11111001111011100101010;
      ternrom[5'b10011]=23'b10100000010101110101011;
      ternrom[5'b10100]=23'b11001101101110110101011;
      ternrom[5'b10101]=23'b10000011111111000101100;
      ternrom[5'b10110]=23'b10101001010110010101100;
      ternrom[5'b10111]=23'b11011001010010011101100;
      ternrom[5'b11000]=23'b10001011011001100101101;
      ternrom[5'b11001]=23'b10110010110111000101101;
      ternrom[5'b11010]=23'b11100101011111100101101;
      ternrom[5'b11011]=23'b10010011001110101101110;
      ternrom[5'b11100]=23'b10111100111010000101110;
      ternrom[5'b11101]=23'b11110010011000100101110;
      ternrom[5'b11110]=23'b10011011011111111101111;
      ternrom[5'b11111]=23'b11000111100001001101111;
  end
  assign {ternaryout,binaryout}=ternrom[ternaryin];

endmodule
```

bshifter.v

This element is a barrel shifter. That means it can shift any number of places in a single clock cycle.

```
module bshifter(ternaryin,shift,binaryout);
      //Default parameters
parameter ltsize=16;
parameter absize=7;
parameter sbsize=15;
      //Define ports
input [ltsize:0] ternaryin;
input [absize:0] shift;
output [sbsize:0] binaryout;

wire [ltsize+2:0] ternarytemp;

      //keep two more bits so the result is more accurate
  assign ternarytemp={ternaryin,2'b00};

      //Perform shift,if positive, shift left
      //if negative, resolve 2's compliment and shift right
      //if zero, do nothing
  assign binaryout =(shift[absize]==0)?(ternarytemp<<shift[absize-
1:0]):(ternarytemp>>((1<<(absize+1))-shift));

endmodule
```

signcorrect.v

This component ensures that the binary output of the DBNS to binary converter has the correct sign corresponding to the multiplication

```
      //Module to correct output sign based on external inputs
module signcorrect (coefsign,datasign,binaryin, binaryout);
      //Default parameters
parameter sbsize=8;
      //Define ports
input [1:0] coefsign;
input [1:0] datasign;
input [sbsize:0] binaryin;
output [sbsize+1:0] binaryout;

      //If the either sign is zero, output should be zero
      //(00 is zero,01 positive,11 negative)
      //If the signs are the same, output the sign extended
      //version of the input
      //If the signs are different, inverse the sign
      //with proper extension
  assign binaryout=(coefsign==0 || datasign ==0) ? 0 : (coefsign != data-
sign ? ((1<<(sbsize+2))-binaryin):{"0",binaryin});

endmodule
```

B.4 Binary to MDLNS converter

These code is written by Mr. Roberto Muscedere

B.4.1 top_convert.v

```
// Define main module
module top_convert( CK,reset,acti-
vate,i0,ready,output_s,output_b,output_t);

// Define clock
input CK;

parameter wsize=16;

parameter isize=17;
parameter ssize=5;

parameter bsize=6;
parameter tsize=5;

parameter csize=2;

parameter mdiff=11;
parameter maxlevel=1;
input [wsize-1:0] i0;
input reset;
input activate;

output ready;
//reg ready;
output [((maxlevel+1)*2)-1:0] output_s;
//reg [((maxlevel+1)*2)-1:0] output_s;
output [(bsize*(maxlevel+1))-1:0] output_b;
//reg [(bsize*(maxlevel+1))-1:0] output_b;
output [(tsize*(maxlevel+1))-1:0] output_t;
//reg [(tsize*(maxlevel+1))-1:0] output_t;

serialdouble #(wsize,isize,ssize,bsize,tsize,csize,mdiff)
serial(CK,reset,activate,i0,ready,output_s,output_b,output_t);

endmodule
```

B.4.2 serialdouble.v

```
// Define Serial Conversion Module
module serialdouble( CK, reset, activate, i, ready, output_s, output_b,
output_t);

// Define parameters
// Input word size in bits
parameter wsize=16;
// Internal working bit size (>= wsize)
parameter isize=17;
// Shifter word size
parameter ssize=5;

// Number of bins in Binary Representation
parameter bsize=6;
// Number of bins in OtherBase Representation
parameter tsize=4;
// Number of bits to omit for comversion (for MAC)
parameter csize=0;
// Internal comparitor accuracy
parameter mdiff=20;

// Define ports
// Data is processed on rising edge
input CK;
input reset;
// Input word in 2's compliment
input [wsize-1:0] i;
// Set to 1 to run converstion on input
input activate;
// Is set to 1 when output data is ready
output ready;
reg ready;
// Output Signs (concatenated)
output [3:0] output_s;
reg [3:0] output_s;
// Output Binary exponent (concatenated)
output [(bsize*2)-1:0] output_b;
reg [(bsize*2)-1:0] output_b;
// Output OtherBase exponent (concatenated)
output [(tsize*2)-1:0] output_t;
reg [(tsize*2)-1:0] output_t;

reg [1:0] final_s[0:1];
reg [bsize-1:0] final_b[0:1];
reg [tsize-1:0] final_t[0:1];

reg [1:0] s_s;
reg [bsize-1:0] s_b[0:1];
reg [tsize-1:0] s_t[0:1];

reg [wsize-1:0] i0;
wire [isize-1:0] i3;
wire [1:0] is3;
separatesign_noclk #(wsize,isize) ss(i0, i3, is3);

reg [isize-1:0] i1;
reg [1:0] is;
```

```
wire [isize-2:0] i1o;
wire [1:0] is2;
wire [ssize:0] shift;
normalizer_noclk #(isize,ssize) no(i1, is, i1o, is2, shift);

reg [isize-2:0] i2;
wire [isize:0] n1,n2;
wire [bsize-1:0] b1,b2;
wire [tsize-1:0] t1,t2;
comparelut_syn #(isize,bsize,tsize,csize) comp(i2,n1,n2,b1,b2,t1,t2);
//comparelut_noclk #(isize,bsize,tsize,csize)
comp(i2,n1,n2,b1,b2,t1,t2);

reg [isize:0] e1,e2,s_e;

reg [ssize:0] d1,d2;
reg [ssize-1:0] h1,h2;
reg [isize-1+mdiff:0] et1,et2;

reg [isize-1:0] best_error;
reg [ssize:0] best_shift,s_shift;

reg [3:0] state;

`ifdef DC
`else
integer max_mdiff;
integer max_ssize;
// Initialization routine (for simulation only)
initial
begin
      max_mdiff=0;
      max_ssize=0;
end
`endif

//synopsys state_vector state
//synopsys sync_set_reset "reset"
always @(posedge CK)
begin

      if (reset)
      begin
// Initialize under reset conditions
            state = 0;
            ready = 0;
            final_s[0] = 0;
            final_b[0] = 0;
            final_t[0] = 0;
            final_s[1] = 0;
            final_b[1] = 0;
            final_t[1] = 0;
      end
      else
```

```
            case (state)
                  0: begin
                        if (activate)
                        begin
// Start conversion
                              state=state+1;
// Load data into sign separator
                              i0 = i;
                              ready = 0;
                              $display("Conversion starts for %d",i);
                        end
                        else
                        begin
// No conversion, continue looping
                              ready = 1;
                        end
                  end

                  1: begin
                        $display("! State: %d",state);

// Get sign and data from sign separator, put into normalizer
                        is = is3;
                        i1 = i3;

                        state=state+1;
                  end

                  2: begin
                        $display("! State: %d",state);
                        if (is2==0 )
                        begin
// input is zero, end conversion
                              final_s[0]=0;
                              final_b[0]=0;
                              final_t[0]=0;
                              final_s[1]=0;
                              final_b[1]=0;
                              final_t[1]=0;

                              ready = 1;
                              state = 0;
                        end
                        else
                        begin
// Get shift and data from normalizer, save
                              final_s[0] = is2;
                              s_shift = shift;
// Load RALUT
                              i2 = i1o;

                              state=state+1;
                        end
```

```
`ifdef DC
`else
                                    if (is2!=0) if (shift>max_ssize)
                                    begin
                                          max_ssize=shift;
                                          $dis-
play("!max_ssize=%d",max_ssize);
                                    end
`endif

                  end

                  3: begin
                        $display("! State: %d",state);
// Load normalizer with error and sign of input
                        e1={2'b01,i2}-n1;
                        i1=e1[wsize-1:0];
                        is=final_s[0];
                        final_b[0]=b1-s_shift;
                        final_t[0]=t1;
                        s_b[0]=b2-s_shift;
                        s_t[0]=t2;
                        s_e=n2-{2'b01,i2};

//    $display("+ i2=%b,n1=%b,e1=%b",i2,n1,e1);
//    $display("+ n2=%b,i2=%b,e2=%b",n2,i2,s_e);

                        state=state+1;
                  end

                  4: begin
                        $display("! State: %d",state);
                        if (is2==0 )
                        begin
// Error is zero, end conversion
                              $display("! second digit sign zero");
                              final_s[1]=0;
                              final_b[1]=0;
                              final_t[1]=0;

                              ready = 1;
                              state = 0;
                        end
                        else
                        begin
// Set output sign
                              final_s[1]=is2;
// Save accumulated shift
                              best_shift=s_shift+shift;
// Load RALUT
                              i2=i1o;
                              state=state+1;
                        end
```

```
`ifdef DC
`else
                                    if (is2!=0) if (shift>max_ssize)
                                    begin
                                          max_ssize=shift;
                                          $dis-
play("!max_ssize=%d",max_ssize);
                                    end
`endif

                  end

                  5: begin
                        $display("! State: %d",state);

                        e1={2'b01,i2}-n1;
                        e2=n2-{2'b01,i2};

                        if ( e1 < e2 )
                        begin
                              final_b[1]=b1-best_shift;
                              final_t[1]=t1;
                              best_error=e1;
                        end
                        else
                        begin
                              final_b[1]=b2-best_shift;
                              final_t[1]=t2;
                              best_error=e2;
                        end

                        $display("! lower : %b %b %b %b %b %b
%b",final_s[0],final_b[0],final_t[0],final_s[1],final_b[1],final_t[1],b
est_error);

// Load normalizer with error and sign of -1
                        i1=s_e[wsize-1:0];
                        if (final_s[0]==1) is=2'b11;
                        else is=2'b01;

                        state=state+1;
                  end


                  6: begin
                        $display("! State: %d",state);
// Save accumulated shift
                        s_shift=s_shift+shift;
                        s_s=is2;
// Load RALUT
                        i2=i1o;
                        state=state+1;

`ifdef DC
```

```
`else
                                    if (is2!=0) if (shift>max_ssize)
                                    begin
                                          max_ssize=shift;
                                          $dis-
play("!max_ssize=%d",max_ssize);
                                    end
`endif

                  end

                  7: begin
                        $display("! State: %d",state);

                        e1={2'b01,i2}-n1;
                        e2=n2-{2'b01,i2};

                        if ( e1 < e2 )
                        begin
                              s_b[1]=b1-s_shift;
                              s_t[1]=t1;
                              s_e=e1;
                        end
                        else
                        begin
                              s_b[1]=b2-s_shift;
                              s_t[1]=t2;
                              s_e=e2;
                        end

                        $display("! higher: %b %b %b %b %b
%b",final_s[0],s_b[0],s_t[0],s_s,s_b[1],s_t[1]);

                        state=state+1;
                  end

                  8: begin
                        $display("! State: %d",state);
//            $display("+ Final: %b,%b",best_error,best_shift);
//            $display("+ Final:
%b,%b",s_i[pointer+1],s_shift[pointer+1]);

                        d1 = s_shift - best_shift;
                        d2 = best_shift - s_shift;

                        if (d1[ssize]==1'b0)
                        begin
// 1 must be shifted
                              h1 = d1;
                              h2 = 0;
                        end
                        else
// 2 must be shifted
                        begin
```

```
                                h1 = 0;
                                h2 = d2;
                        end

                        state=state+1;

// Test stuff
`ifdef DC
`else
                                        if (best_shift !=0)
                                        begin
                                                if (h1>max_mdiff)
                                                begin
                                                max_mdiff=h1;
                                                $dis-
play("!max_mdiff=%d",max_mdiff);
                                                end
                                                if (h2>max_mdiff)
                                                begin
                                                max_mdiff=h2;
                                                $dis-
play("!max_mdiff=%d",max_mdiff);
                                                end
                                        end
`endif

                end

                9: begin
                        $display("! State: %d",state);
                        et1 = ( s_e << mdiff ) >> h1;
                        et2 = ( best_error << mdiff ) >> h2;

//      $display("R1: %b %b %b %b",et1,d1,h1,s_shift);
//      $display("R2: %b %b %b %b",et2,d2,h2,best_shift);

                        if (et1 < et2)
                        begin
//                              $display("Using higher");
                                final_b[0]=s_b[0];
                                final_t[0]=s_t[0];
                                final_s[1]=s_s;
                                final_b[1]=s_b[1];
                                final_t[1]=s_t[1];
                        end

// Conversion complete
                        ready = 1;
                        state = 0;
                end

                default: begin
                        state = 0;
                        ready = 0;
```

```
                  end

            endcase


end // always

`ifdef DC
always
`else
always @(posedge ready)
`endif
begin
      output_s = {final_s[1],final_s[0]};
      output_b = {final_b[1],final_b[0]};
      output_t = {final_t[1],final_t[0]};
end

endmodule
```

B.4.3 separatesign_noclk.v

```
// Module for separating an integer into a number and a sign
//  This module doesn't generate a zero sign, simply a 1 or -1
module separatesign_noclk (i, o, os);

// Default parameters
// Input word size in bits
parameter isize = 16;
// Output word size in bits (must be equal to or greater than isize)
parameter osize = 15;

// Define ports
// Input of module in 2's compliments form
input [isize-1:0] i;
// Output number of module in binary form
output [osize-1:0] o;
reg [osize-1:0] o;
// Output sign of module
output [1:0] os;
reg [1:0] os;

`ifdef DC
`else
initial
begin
      // Stop simulation if output is smaller than input
      if (osize<isize) $stop;
end
`endif

// Two intermediate registers to expand word length
reg [31:0] t1,t2;
```

```
always @(i)
begin

      // Generate negative value of input, but extend the sign too
      t1 = 0 - { {(31-isize){i[isize-1]}}, i[isize-2:0]};
      // Copy input
      t2 = i;

      // Check the high bit of the input, if it is one, then input
      //  is negative
      if (i[isize-1]==1'b1)
      begin
            // Set output to negated input
            o = t1[osize-1:0];
            // Set sign to -1
            os = 2'b11;
      end
      else
      begin
            // Set output to extended input
            o = t2[osize-1:0];
            // Set sign to 1
            os = 2'b01;
      end

//    $display("- Separatesign: %b %b %b",i,o,os);
end

endmodule
```

B.4.4 normalizer_noclk.v

```
// Module for normalizing and integer and possibly setting the sign to
zero
module normalizer_noclk (i, is, o, os, s);

// Default parameters
// isize is the input number of bits, the output will be isize-1 bits
since
//  the first "1" will be omitted
parameter isize = 16;
// ssize is the number of bits to describe the shift of the normalization
parameter ssize = 4;

// Define ports
// input
input [isize-1:0] i;
// input sign
input [1:0] is;
// output of normalization,
output [isize-2:0] o;
reg [isize-2:0] o;
// output sign, will be zero is input sign is also
```

```
output [1:0] os;
reg [1:0] os;
// output shift
output [ssize:0] s;
reg [ssize:0] s;

// temporary variables
reg [isize-1:0] t;
reg z;

// internal counter
integer c;

always @(i or is)
begin

      // there are other ways to do this procedure, but through synthesis
      //  this turns out to be the smallest

      // set shift to zero
      s=0;
      // set loop control
      z=0;

      // loop through all bits from left to right
      for (c=isize-1;c>=0;c=c-1)
      begin
            // if the bit is zero and no ones have been encountered,
            //  keep updating this shift
            if (i[c]==1'b0 && z==1'b0)
            begin
                  // record the size
                  s=isize - c;
                  // set the loop to keep going
                  z=0;
            end
            else
            begin
                  // don't infer any latches
                  s=s;
                  // a one has been spotted, no more updates
                  z=1;
            end
      end

      // shift temporary register
      t = i << s;
      // set output from temporary register (we can not do this in one
step)
      o = t[isize-2:0];

      // check the high bit of the temporary register
      if (t[isize-1]==1'b0)
      begin
```

```
            // if zero, set the output size to zero
            os=0;
      end
      else
      begin
            // if not, pass the input sign through
            os=is;
      end

//    $display("- Normalizer: %b %b %b %b %b",i,is,o,os,s);
end

endmodule
```

B.4.5 comparelut_syn.v

```
module comparelut_syn (i, on, onp, ob, obp, ot, otp);

// Default parameters
parameter isize = 17;
parameter bsize = 6;
parameter tsize = 4;
parameter csize = 0;

parameter mx = (1 << tsize) - (1<< csize) + 1;
parameter mxm = mx-1;

// Define ports
input [isize-2:0] i;
output [isize:0] on,onp;
reg [isize:0] on,onp;
output [bsize-1:0] ob,obp;
reg [bsize-1:0] ob,obp;
output [tsize-1:0] ot,otp;
reg [tsize-1:0] ot,otp;

reg [mxm:0] d;

reg [mxm:0] c;

reg [isize-2:0] romA[mxm:0];
reg [isize-2:0] romB[mxm:0];
reg [tsize-1:0] romC[mxm:0];
reg [tsize-1:0] romD[mxm:0];
reg [bsize-1:0] romE[mxm:0];
reg [bsize-1:0] romF[mxm:0];

reg [30:0] high;

wire [isize-2:0] fin_on,fin_onp;
reg [isize-2:0] tri_on[mxm:0],tri_onp[mxm:0];
wire [bsize-1:0] fin_ob,fin_obp;
reg [bsize-1:0] tri_ob[mxm:0],tri_obp[mxm:0];
```

```
wire [tsize-1:0] fin_ot,fin_otp;
reg [tsize-1:0] tri_ot[mxm:0],tri_otp[mxm:0];

integer p;
integer j;
integer k;

// nosynopsys infer_multibit "fin_on"
'include "comparelut_temp2.v"

'ifdef DC
'else
initial
begin
      'include "comparelut_temp.v"
      high = 31'bz;
      c[mx]=0;
end
'endif

always @(i)
begin
'ifdef DC
      'include "comparelut_temp.v"
      high = 31'bz;
'else
'endif

      for (p=0;p<=mxm;p=p+1)
      begin
            if (p==mxm) c[p] = (i>=romA[p]);
            else c[p] = (i>=romA[p] && i<romA[p+1]);
      end

      for (k=0;k<=mxm;k=k+1)
      begin
            if (c[k])
            begin
                  tri_on[k] = romA[k];
                  tri_onp[k] = romB[k];
                  tri_ot[k] = romC[k];
                  tri_otp[k] = romD[k];
                  tri_ob[k] = romE[k];
                  tri_obp[k] = romF[k];
            end
            else
            begin
                  tri_on[k] = high[30:30-isize+2];
                  tri_onp[k] = high[30:30-isize+2];
                  tri_ob[k] = high[bsize-1:0];
                  tri_obp[k] = high[bsize-1:0];
                  tri_ot[k] = high[tsize-1:0];
                  tri_otp[k] = high[tsize-1:0];
            end
      end
end

'ifdef DC
always
'else
always @(fin_on or fin_onp or fin_ob or fin_obp or fin_ot or fin_otp)
'endif
begin
      on[isize-2:0] = fin_on;
      on[isize:isize-1] = 2'b01;
      onp[isize-2:0] = fin_onp;
      onp[isize:isize] = c[mxm];
      onp[isize-1:isize-1] = ! c[mxm];

      ob[bsize-1:0] = fin_ob;
      obp[bsize-1:0] = fin_obp;

      ot[tsize-1:0] = fin_ot;
      otp[tsize-1:0] = fin_otp;

end

endmodule
```

B.4.6 comparelut_temp.v

```
romA[0]=16'b0000000000000000;romC[0]=0;romE[0]=16;romB[0]=16'b0000011000110111;romD[0]=1
romF[0]=11;
romA[1]=16'b0000011000110111;romC[1]=14;romE[1]=11;romB[1]=16'b0000001111111000;
romD[1]=-11;romF[1]=20;
romA[2]=16'b0000001111111000;romC[2]=-11;romE[2]=20;romB[2]=16'b0000111001100001;romD[2]=3
romF[2]=15;
romA[3]=16'b0000111001100001;romC[3]=3;romE[3]=15;romB[3]=16'b0001011011001100;
romD[3]=-8;romF[3]=19;
romA[4]=16'b0001011011001100;romC[4]=-8;romE[4]=19;romB[4]=16'b0001110110010001;romD[4]=6
romF[4]=14;
romA[5]=16'b0001110110010001;romC[5]=6;romE[5]=14;romB[5]=16'b0010011001110101;
romD[5]=-5;romF[5]=18;
romA[6]=16'b0010011001110101;romC[6]=-5;romE[6]=18;romB[6]=16'b0010110110011011;romD[6]=9
romF[6]=13;
romA[7]=16'b0010110110011011;romC[7]=9;romE[7]=13;romB[7]=16'b0011011011111111;
romD[7]=-2;romF[7]=17;
romA[8]=16'b0011011011111111;romC[8]=-2;romE[8]=17;romB[8]=16'b0011111010001101;romD[8]=1
romF[8]=12;
romA[9]=16'b0011111010001101;romC[9]=12;romE[9]=12;romB[9]=16'b0100000010101110;
romD[9]=-13;romF[9]=21;
romA[10]=16'b0100000010101110;romC[10]=-13;romE[10]=21;romB[10]=16'b0100100001111000;romD[10]=
romF[10]=16;
romA[11]=16'b0100100001111000;romC[11]=1;romE[11]=16;romB[11]=16'b0101001010110010;
romD[11]=-10;romF[11]=20;
```

```
romA[12]=16'b0101001010110010;romC[12]=-10;romE[12]=20;romB[12]=16'b0101101011101011;romD[12]=4
romF[12]=15;
romA[13]=16'b0101101011101011;romC[13]=4;romE[13]=15;romB[13]=16'b0110010110111000;
romD[13]=-7;romF[13]=19;
romA[14]=16'b0110010110111000;romC[14]=-7;romE[14]=19;romB[14]=16'b0110111001101000;romD[14]=
romF[14]=14;
romA[15]=16'b0110111001101000;romC[15]=7;romE[15]=14;romB[15]=16'b0111100111010000;
romD[15]=-4;romF[15]=18;
romA[16]=16'b0111100111010000;romC[16]=-4;romE[16]=18;romB[16]=16'b1000001011111101;romD[16]=
romF[16]=13;
romA[17]=16'b1000001011111101;romC[17]=10;romE[17]=13;romB[17]=16'b1000111100001001;
romD[17]=-1;romF[17]=17;
romA[18]=16'b1000111100001001;romC[18]=-1;romE[18]=17;romB[18]=16'b1001100010111010;romD[18]=
romF[18]=12;
romA[19]=16'b1001100010111010;romC[19]=13;romE[19]=12;romB[19]=16'b1001101101110110;
romD[19]=-12;romF[19]=21;
romA[20]=16'b1001101101110110;romC[20]=-12;romE[20]=21;romB[20]=16'b1010010101110100;romD[20]=
romF[20]=16;
romA[21]=16'b1010010101110100;romC[21]=2;romE[21]=16;romB[21]=16'b1011001010010011;
romD[21]=-9;romF[21]=20;
romA[22]=16'b1011001010010011;romC[22]=-9;romE[22]=20;romB[22]=16'b1011110100100000;romD[22]=
romF[22]=15;
romA[23]=16'b1011110100100000;romC[23]=5;romE[23]=15;romB[23]=16'b1100101011111100;
romD[23]=-6;romF[23]=19;
romA[24]=16'b1100101011111100;romC[24]=-6;romE[24]=19;romB[24]=16'b1101011000100001;romD[24]=
romF[24]=14;
romA[25]=16'b1101011000100001;romC[25]=8;romE[25]=14;romB[25]=16'b1110010011000100;
romD[25]=-3;romF[25]=18;
romA[26]=16'b1110010011000100;romC[26]=-3;romE[26]=18;romB[26]=16'b1111000010001010;romD[26]=
romF[26]=13;
romA[27]=16'b1111000010001010;romC[27]=11;romE[27]=13;romB[27]=16'b1111001111011100;
romD[27]=-14;romF[27]=22;
romA[28]=16'b1111001111011100;romC[28]=-14;romE[28]=22;romB[28]=16'b0000000000000000;romD[28]=
romF[28]=17;
```

B.4.7 comparelut_temp2.v

```
assign fin_on = tri_on[0], fin_onp = tri_onp[0], fin_ob = tri_ob[0],
fin_obp = tri_obp[0], fin_ot = tri_ot[0], fin_otp = tri_otp[0];
assign fin_on = tri_on[1], fin_onp = tri_onp[1], fin_ob = tri_ob[1],
fin_obp = tri_obp[1], fin_ot = tri_ot[1], fin_otp = tri_otp[1];
assign fin_on = tri_on[2], fin_onp = tri_onp[2], fin_ob = tri_ob[2],
fin_obp = tri_obp[2], fin_ot = tri_ot[2], fin_otp = tri_otp[2];
assign fin_on = tri_on[3], fin_onp = tri_onp[3], fin_ob = tri_ob[3],
fin_obp = tri_obp[3], fin_ot = tri_ot[3], fin_otp = tri_otp[3];
assign fin_on = tri_on[4], fin_onp = tri_onp[4], fin_ob = tri_ob[4],
fin_obp = tri_obp[4], fin_ot = tri_ot[4], fin_otp = tri_otp[4];
assign fin_on = tri_on[5], fin_onp = tri_onp[5], fin_ob = tri_ob[5],
fin_obp = tri_obp[5], fin_ot = tri_ot[5], fin_otp = tri_otp[5];
assign fin_on = tri_on[6], fin_onp = tri_onp[6], fin_ob = tri_ob[6],
fin_obp = tri_obp[6], fin_ot = tri_ot[6], fin_otp = tri_otp[6];
assign fin_on = tri_on[7], fin_onp = tri_onp[7], fin_ob = tri_ob[7],
fin_obp = tri_obp[7], fin_ot = tri_ot[7], fin_otp = tri_otp[7];
```

```
assign fin_on = tri_on[8], fin_onp = tri_onp[8], fin_ob = tri_ob[8],
fin_obp = tri_obp[8], fin_ot = tri_ot[8], fin_otp = tri_otp[8];
assign fin_on = tri_on[9], fin_onp = tri_onp[9], fin_ob = tri_ob[9],
fin_obp = tri_obp[9], fin_ot = tri_ot[9], fin_otp = tri_otp[9];
assign fin_on = tri_on[10], fin_onp = tri_onp[10], fin_ob = tri_ob[10],
fin_obp = tri_obp[10], fin_ot = tri_ot[10], fin_otp = tri_otp[10];
assign fin_on = tri_on[11], fin_onp = tri_onp[11], fin_ob = tri_ob[11],
fin_obp = tri_obp[11], fin_ot = tri_ot[11], fin_otp = tri_otp[11];
assign fin_on = tri_on[12], fin_onp = tri_onp[12], fin_ob = tri_ob[12],
fin_obp = tri_obp[12], fin_ot = tri_ot[12], fin_otp = tri_otp[12];
assign fin_on = tri_on[13], fin_onp = tri_onp[13], fin_ob = tri_ob[13],
fin_obp = tri_obp[13], fin_ot = tri_ot[13], fin_otp = tri_otp[13];
assign fin_on = tri_on[14], fin_onp = tri_onp[14], fin_ob = tri_ob[14],
fin_obp = tri_obp[14], fin_ot = tri_ot[14], fin_otp = tri_otp[14];
assign fin_on = tri_on[15], fin_onp = tri_onp[15], fin_ob = tri_ob[15],
fin_obp = tri_obp[15], fin_ot = tri_ot[15], fin_otp = tri_otp[15];
assign fin_on = tri_on[16], fin_onp = tri_onp[16], fin_ob = tri_ob[16],
fin_obp = tri_obp[16], fin_ot = tri_ot[16], fin_otp = tri_otp[16];
assign fin_on = tri_on[17], fin_onp = tri_onp[17], fin_ob = tri_ob[17],
fin_obp = tri_obp[17], fin_ot = tri_ot[17], fin_otp = tri_otp[17];
assign fin_on = tri_on[18], fin_onp = tri_onp[18], fin_ob = tri_ob[18],
fin_obp = tri_obp[18], fin_ot = tri_ot[18], fin_otp = tri_otp[18];
assign fin_on = tri_on[19], fin_onp = tri_onp[19], fin_ob = tri_ob[19],
fin_obp = tri_obp[19], fin_ot = tri_ot[19], fin_otp = tri_otp[19];
assign fin_on = tri_on[20], fin_onp = tri_onp[20], fin_ob = tri_ob[20],
fin_obp = tri_obp[20], fin_ot = tri_ot[20], fin_otp = tri_otp[20];
assign fin_on = tri_on[21], fin_onp = tri_onp[21], fin_ob = tri_ob[21],
fin_obp = tri_obp[21], fin_ot = tri_ot[21], fin_otp = tri_otp[21];
assign fin_on = tri_on[22], fin_onp = tri_onp[22], fin_ob = tri_ob[22],
fin_obp = tri_obp[22], fin_ot = tri_ot[22], fin_otp = tri_otp[22];
assign fin_on = tri_on[23], fin_onp = tri_onp[23], fin_ob = tri_ob[23],
fin_obp = tri_obp[23], fin_ot = tri_ot[23], fin_otp = tri_otp[23];
assign fin_on = tri_on[24], fin_onp = tri_onp[24], fin_ob = tri_ob[24],
fin_obp = tri_obp[24], fin_ot = tri_ot[24], fin_otp = tri_otp[24];
assign fin_on = tri_on[25], fin_onp = tri_onp[25], fin_ob = tri_ob[25],
fin_obp = tri_obp[25], fin_ot = tri_ot[25], fin_otp = tri_otp[25];
assign fin_on = tri_on[26], fin_onp = tri_onp[26], fin_ob = tri_ob[26],
fin_obp = tri_obp[26], fin_ot = tri_ot[26], fin_otp = tri_otp[26];
assign fin_on = tri_on[27], fin_onp = tri_onp[27], fin_ob = tri_ob[27],
fin_obp = tri_obp[27], fin_ot = tri_ot[27], fin_otp = tri_otp[27];
assign fin_on = tri_on[28], fin_onp = tri_onp[28], fin_ob = tri_ob[28],
fin_obp = tri_obp[28], fin_ot = tri_ot[28], fin_otp = tri_otp[28];
```

Appendix C

Matlab Code for MDLNS and Binary Filterbank Comparison

C.1 Introduction

These files are used to compare the verilog simulation output of the MDLNS filterbank and the output of its 16-bit binary counterpart.

C.2 floating_output.m

Computed floating point output of the filterbank

```
        %load input chirp signal
load inputchirpDEC9000.txt
         %load filters coefficients
load b1; %from Appendix A
load b2;
load b3;
load b4;
load b5;
load b6;
load b7;
load b8;
% Reset coefficient RAM
RAM=zeros(75,1);
% computer outputs of the filterbank
for n=1:18500
    temp=RAM;
```

```
        for i=2:75
        RAM(i)=temp(i-1);
        end
      RAM(1)=inputchirpDEC9000(n);
       out1(n)=sum((RAM(1:74).').*b1);
       out2(n)=sum((RAM(1:74).').*b2);
       out3(n)=sum((RAM(1:74).').*b3);
       out4(n)=sum((RAM(1:74).').*b4);
       out5(n)=sum((RAM(1:74).').*b5);
       out6(n)=sum((RAM(1:74).').*b6);
       out7(n)=sum((RAM(1:74).').*b7);
       out8(n)=sum((RAM.').*b8);
end
% normalized the filter output
 float8output =[out1/max(abs(out1)); out2/
max(abs(out2)); out3/max(abs(out3)); out4/
max(abs(out4)); out5/max(abs(out5)); out6/
max(abs(out6)); out7/max(abs(out7)); out8/
max(abs(out8))];
   save float8output float8output;
for i=1:8
   subplot(8,1,i),plot(float8output(i,:));
end
```

C.3 binary_output.m

Computed 16-bit binary quantized filterbank output

```
load inputchirpDEC9000.txt; % load input chirp signal
load b1;
load b2;
load b3;
load b4;
load b5;
load b6;
load b7;
load b8;
```

```
q=quantizer('fixed','round','saturate',[16 15]);
q1=quantizer('fixed','round','saturate',[16 14]);

% quantized the coefficients
qb1=quantize(q,b1);
qb2=quantize(q,b2);
qb3=quantize(q,b3);
qb4=quantize(q,b4);
qb5=quantize(q,b5);
qb6=quantize(q,b6);
qb7=quantize(q,b7);
qb8=quantize(q,b8);

% quantized the input chirp signal
qinputchirpDEC=quantize(q,inputchirpDEC9000);
RAM=zeros(75,1); % Reset coefficient RAM

% computer filterbank outputs
for n=1:18500
     temp=RAM;
            for i=2:75
                RAM(i)=temp(i-1);
            end
     RAM(1)=qinputchirpDEC(n);
     out1(n)=sum(quantize(q1,(RAM(1:74).').*b1));
     out2(n)=sum(quantize(q1,(RAM(1:74).').*b2));
     out3(n)=sum(quantize(q1,(RAM(1:74).').*b3));
     out4(n)=sum(quantize(q1,(RAM(1:74).').*b4));
     out5(n)=sum(quantize(q1,(RAM(1:74).').*b5));
     out6(n)=sum(quantize(q1,(RAM(1:74).').*b6));
     out7(n)=sum(quantize(q1,(RAM(1:74).').*b7));
     out8(n)=sum(quantize(q1,(RAM.').*b8));
end

% quantized the output
qout1=quantize(q1,out1);
qout2=quantize(q1,out2);
qout3=quantize(q1,out3);
qout4=quantize(q1,out4);
```

```
qout5=quantize(q1,out5);
qout6=quantize(q1,out6);
qout7=quantize(q1,out7);
qout8=quantize(q1,out8);

% normalized the output
qoutnom1=qout1/max(abs(qout1));
qoutnom2=qout2/max(abs(qout2));
qoutnom3=qout3/max(abs(qout3));
qoutnom4=qout4/max(abs(qout4));
qoutnom5=qout5/max(abs(qout5));
qoutnom6=qout6/max(abs(qout6));
qoutnom7=qout7/max(abs(qout7));
qoutnom8=qout8/max(abs(qout8));
save quan16output qoutnom1 qoutnom2 qoutnom3 qoutnom4
qoutnom5 qoutnom6 qoutnom7 qoutnom8

x=0:0.4865:9000;
 subplot(9,1,1),plot(x,inputchirpDEC9000);
title('input chirp wave from 1Hz to 9000Hz');
subplot(9,1,2),plot(x,qoutnom1);
subplot(9,1,3),plot(x,qoutnom2)
subplot(9,1,4), plot(x,qoutnom3)
subplot(9,1,5), plot(x,qoutnom4)
subplot(9,1,6),plot(x,qoutnom5)
ylabel('Magnitude');
subplot(9,1,7),plot(x,qoutnom6)
subplot(9,1,8),plot(x,qoutnom7)
subplot(9,1,9),plot(x,qoutnom8)
xlabel('Frequence in Hz');
```

C.4 noise_bin_floating.m

Noise between 16-bit binary and floating point filterbank outputs

```
load float8output;
load quan16output;
x=0:0.4865:9000;
qoutnom=[qoutnom1; qoutnom2; qoutnom3; qoutnom4;
qoutnom5; qoutnom6; qoutnom7; qoutnom8];

errorbf=qoutnom-float8output;
title('input chirp wave from 1Hz to 9000Hz');
for i=1:8
figure
 plot(x,errorbf(i,:));
end
```

C.5 noise_MDLNS_floating.m

Noise between the MDLNS quantized output and floating point output of the filterbank

```
load float8output;
load MDLNSout1.txt;    %load verilog simulation output
load MDLNSout2.txt;    % of the MDLNS filterbank
load MDLNSout3.txt;
load MDLNSout4.txt;
load MDLNSout5.txt;
load MDLNSout6.txt;
load MDLNSout7.txt;
load MDLNSout8.txt;
MDLNSoutnom1=MDLNSout1/max(abs(MDLNSout1));
MDLNSoutnom2=MDLNSout2/max(abs(MDLNSout2));
MDLNSoutnom3=MDLNSout3/max(abs(MDLNSout3));
MDLNSoutnom4=MDLNSout4/max(abs(MDLNSout4));
MDLNSoutnom5=MDLNSout5/max(abs(MDLNSout5));
MDLNSoutnom6=MDLNSout6/max(abs(MDLNSout6));
MDLNSoutnom7=MDLNSout7/max(abs(MDLNSout7));
MDLNSoutnom8=MDLNSout8/max(abs(MDLNSout8));

MDLNSoutnom=[MDLNSoutnom1 MDLNSoutnom2 MDLNSoutnom3
MDLNSoutnom4 MDLNSoutnom5 MDLNSoutnom6 MDLNSoutnom7
MDLNSoutnom8];
```

```
MDLNSoutnom=MDLNSoutnom';


x=0:0.4865:9000;
for i=1:8
error(i,:)=MDLNSoutnom(i,:)-float8output(i,1:18500);
 figure
 plot(x,(error(i,:)));
end
```

Appendix D
MDLNS Filterbank Chip Testing

D.1 Test Overview

The first step of the test is to take the functional test vectors that were used during the filterbank chip Verilog simulation and translate them into a PCF file. The test vectors then are imported into the D20 digital test system. This requires setting up the software component of the tester to accept the vectors and creating the associated timing parameters. The hardware must be set up and the Device Under Test (DUT) is fixtured to the tester. Then the functional test is run to determine if the DUT is functioning as designed. Finally, test program for current measurement is written with HP's VEE Test in order to estimate the power consumption.

The hardware required for the test:

a) CMC's TH1000 Test Head
b) CMC's VXIbus Digital Test System:
i) HP 745i Workstation
ii) VXIbus E1401 Main Frame with the following modules:
 HP E1406A Command Module
 HP E1450A 160MHz Timing Module
 HP E1451A 20MHz Pattern I/O Modules
 HP E1452A 20MHz Terminating Pattern I/O Module

The software required for the test:

a) HP 75000 Model D20 HP 1496A Digital Test Development software

b) HP VEE Test Software

Figure D.1 and Figure D.2 are the test setup for MDLNS filterbank chip testing

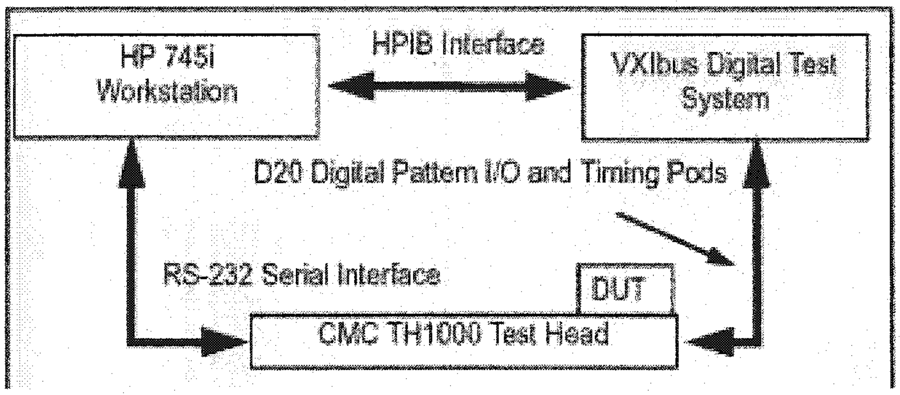


Figure D.1 Overview of the test setup [27]

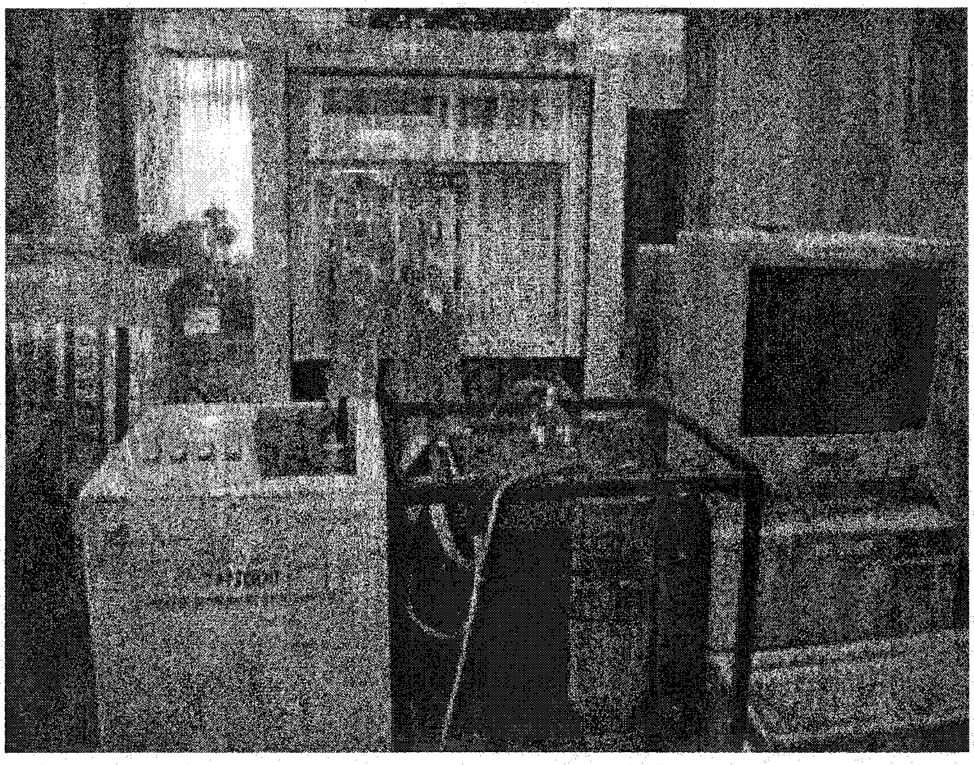


Figure D.2 Testing lab for MDLNS Filterbank Processor Chip

D.2 Test Procedure

D.2.1 Test Vector Generation

Pattern Capture Format (PCF) is a small subset of the Vector Control Language and is itself a specific format used to define test vectors. It is an ASCII file containing vector data which can be imported into the digital test system. We run an AWK script (There is an example on [27])to translate the simulation test vector file to the D20-ready PCF file filterbankchip.pcf which is shown in following. There are total 65535 test vectors in the test.

```
pcf order is input, output, ready_start
pcf
!input output ready_start
use timing set Write_Read
"010 LL LL"
"110 LL LL"
"110 LL LL"
"110 LL LL"
"110 LL LL"
"110 LL LL"
"110 LL LL"
"110 LL LL"
      .
      .
      .
      .
"101 HH HH"
"101 LH HH"
"101 LH HH"
"101 LH LL"
"101 LH LL"
"101 LH LL"
"101 LH LL"
"101 LH LL"
"101 LH LL"
"101 LH LL"
"101 LH LL"
end pcf
```

D.2.2 D20 E1496A Developmental Software

To start the D20 software, login to the HP 745i workstation (scry.vlsi.uwindsor.ca) and `type` **hp75000d20.** This will bring up the D20 digital test system software consisting of a vector spreadsheet and a timing cycle worksheet. The first thing that must be done in order to import the PCF file is to define the pin groups. Pin groups are user-named groupings of tester pins that can either be Control Output or Pattern I/O.

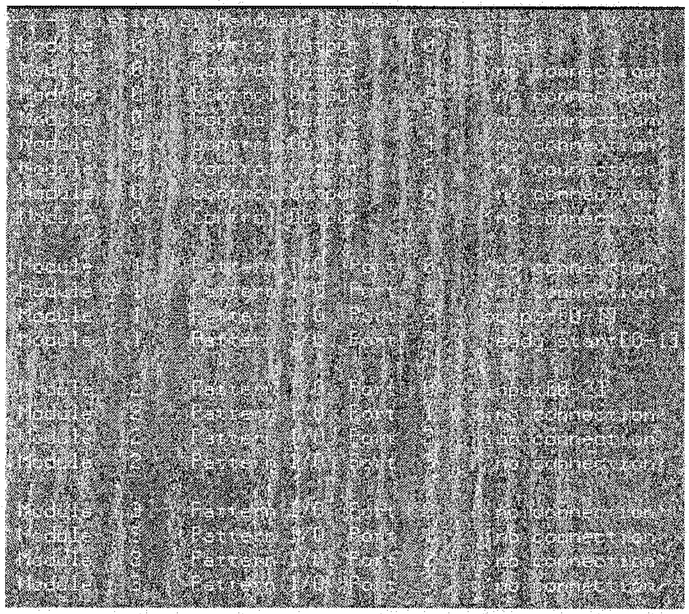
Pin Group Definition

For Pattern I/O each pin group contains one to 32 hardware pins: each HP E1451/E1452 Pattern I/O VXI Card is a Module, each Module has four Ports (two pods/module) and each Port has eight dedicated Pattern I/O pins. Each pin group must have a unique name, be defined as either Stimulus or Response, and must be assigned a timing clock. Timing clocks are used to synchronize pin group clocks [27].

we define three input pins of the MDLNS filterbank chip "reset", "enable" and "inputbit" as one pin group "input", which is **Pattern I/O** type and **Stimulus** mode, two output pins "sout1", "sout2" as one pin group "output" and two output pins "start", "ready" as one pin group "ready_start", which are **Pattern I/O** type and **Compare** mode.

Choosing **Compare** instead of **Record** for the Mode of this pin grouping causes the D20 to read the data from the Device Under Test and compare it against expected values. The expected values are contained in the PCF file.

Control Output pin groups are pin groups which are defined per pin and are lines from the Control Output Port on the HP E1450 Timing Module. We define one pin "CK" as one Control Output type pin group "clock"



```
[illegible] Listing of Hardware Connections
Module  [illegible]  Control Output  [illegible]  clock
Module  [illegible]  Control Output  [illegible]  no connection
Module  [illegible]  Control Output  [illegible]  no connection
Module  [illegible]  Control Output  [illegible]  no connection
Module  [illegible]  Control Output  [illegible]  no connection
Module  [illegible]  Control Output  [illegible]  no connection
Module  [illegible]  Control Output  [illegible]  no connection
Module  [illegible]  Control Output  [illegible]  no connection

Module  1   Pattern I/O Port 0   no connection
Module  1   Pattern I/O Port 1   no connection
Module  1   Pattern I/O Port 2   output[0-N]
Module  1   Pattern I/O Port 3   ready_start[0-1]

Module  2   Pattern I/O Port 0   input[0-2]
Module  2   Pattern I/O Port 1   no connection
Module  2   Pattern I/O Port 2   no connection
Module  2   Pattern I/O Port 3   no connection

Module  3   Pattern I/O Port 0   no connection
Module  3   Pattern I/O Port 1   no connection
Module  3   Pattern I/O Port 2   no connection
Module  3   Pattern I/O Port 3   no connection
```

Figure D.3 List of hardware connections

Figure D.3 shows the list all of the defined pin groups and their hardware equivalent. We will fixture the DUT to the tester according to this list.

Defining Timing Cycles

Setting up the Timing Cycles is similar to graphically creating a timing diagram for the signals. See *HP 75000 Model D20 Reference Manual* for a list of the rules that apply to the Creation of Timing Cycles. Figure D.4 shows the timing cycle of MDLNS filterbank chip test.

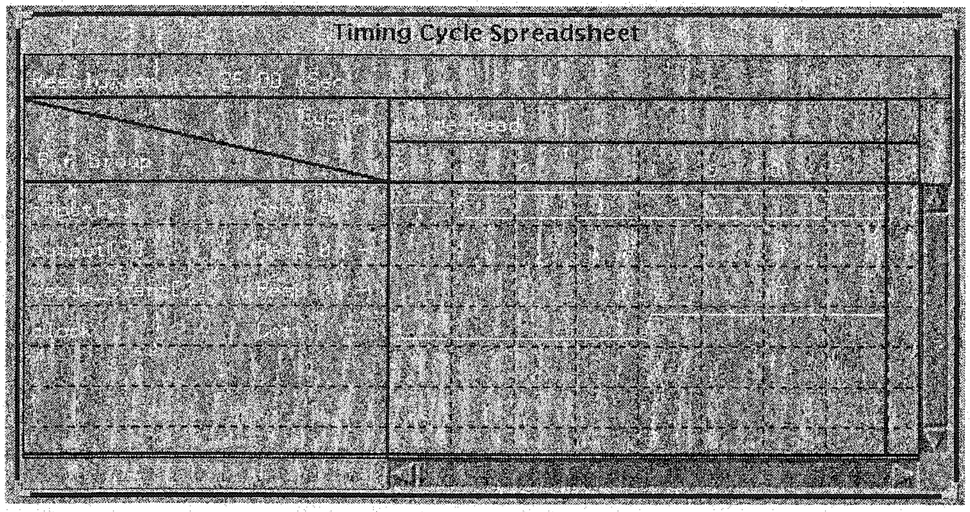


Figure D.4 Timing cycle of MDLNS filterbank chip testing

In summary the timing resolution is 25nSec, subcycle number is 8. Timing cycle is 200nSec long, therefore new test vectors will be sent to this DUT at a 5MHz data rate. This data will change on subcycle 1, be clocked into the device on subcycle 4 and sampled at the output on subcycle 6.

D.2.3 Fixturing the Device Under Test

One of the most time-consuming tasks involving the most probable sources of error when testing a device is the fixturing. If the design is reasonably complicated then it is quite easy to miss a connection or inadvertently connect the wrong signals together.

The CMC IC Test Head (TH1000) is a low cost test fixture designed to support the testing of mixed-signal integrated circuits. Up to 256 connections can be made to the Device Under Test from a variety of sources including fixed and programmable power supplies, Stimulus and Measurement Units (SMUs) and external analog and digital signals. A standard RS-232 interface is the primary communication link with the Test Head [27].

The TH1000 system includes a general purpose Pin Grid Array (PGA) DUT board with a zero insertion force (ZIF) socket which will accept any PGA package up to 16 x 16 pins in size. The board consists of three regions: the PGA Socket, the Inner Wiring Array (IWA) and the Outer Wiring Array (see Figure D.5).

The TH1000 includes three fixed power supplies (+5V, +15V, and -15V) and five programmable power supplies. The outputs of these power supplies are available from posts on the top of the TH1000. The fixed power supplies have 3% accuracy and short circuit protection The five programmable power supplies have +/- 0.1% regulation accuracy in both current/voltage and a minimum setpoint resolution of 12-bits or 5 mV and 73 uA [27].

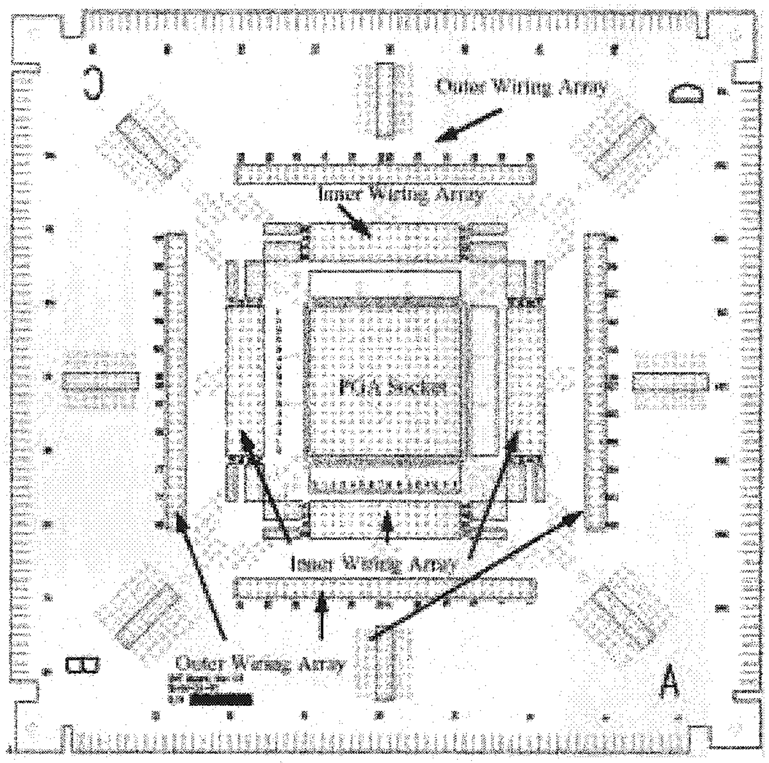


Figure D.5 Device Under Test Board

The package of MDLNS filterbank processor chip is 40 DIP (2 X 20), which we can't direct fit into DUT PGA socket (16 X 16). We use a pin converter to place the chip into the PGA socket. According to pin group and hardware connection list shown in Figure D.3, we made a pin-to-pod mapping table (Table D.1) to ensure that all the pins on the IC (except Power and Ground pins) are mapping to signals on D20 Pods. By using jumper wires and wire wrap, we run the interconnect wiring from the IWA to the OWA according Table D.1. There are 8 anxiliary connectors on the outside of the OWA. We use these

Table D.1 pin-to-pod mapping

IC	signal name	IWA	OWA	POD_##
2	CK (input)	0E	7	0:0 0
9	sout2 (output)	8E	224	1:2 0
12	sout1 (output)	1F	225	1:2 1
19	start (output)	8F	240	1:3 0
22	ready (output)	81	241	1:3 1
29	enable (input)	11	193	2:0 1
32	datain (input)	82	192	2:0 0
39	reset (input)	12	194	2:0 3

auxiliary connectors as our power supply ring busses by soldering them.

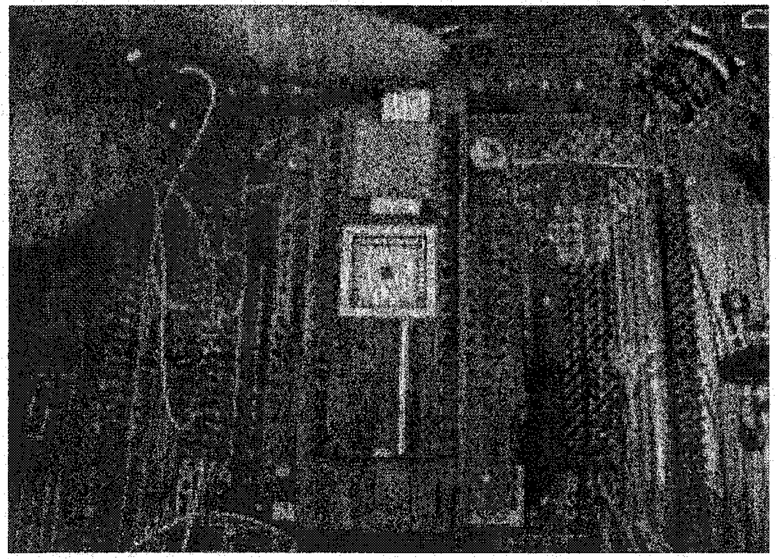


Figure D.6 Testing chip and DUT board

Once the chip is fixtured onto the board as shown in Figure D.6, we install the DUT board on the TH1000 Test Head, then connect interfaces to the Pod of the HP E1450 timing module and HP E1451/1452 I/O modules according to the hardware connection list.

Two separate power supply needed for ring power pads and core pads of the DUT are 3.3v and 1.8 v respectively.

When we started the D20 software, filterbankchip`.pcf` file that was created in Section D.2.1 was loaded first. Then we ran the test from the main menu by selecting **Debug -> Run Test**. This will allow us to run the functional test interactively with the D20 Hardware, once the faulty response is encountered it will be displayed in a message window giving the test vector number, expected value.

D.2.4 HP VEE Test Software

HP VEE (Visual Engineering Environment) is a graphical programming language where programs are created by connecting icons (objects) together. Each object performs a specific task .

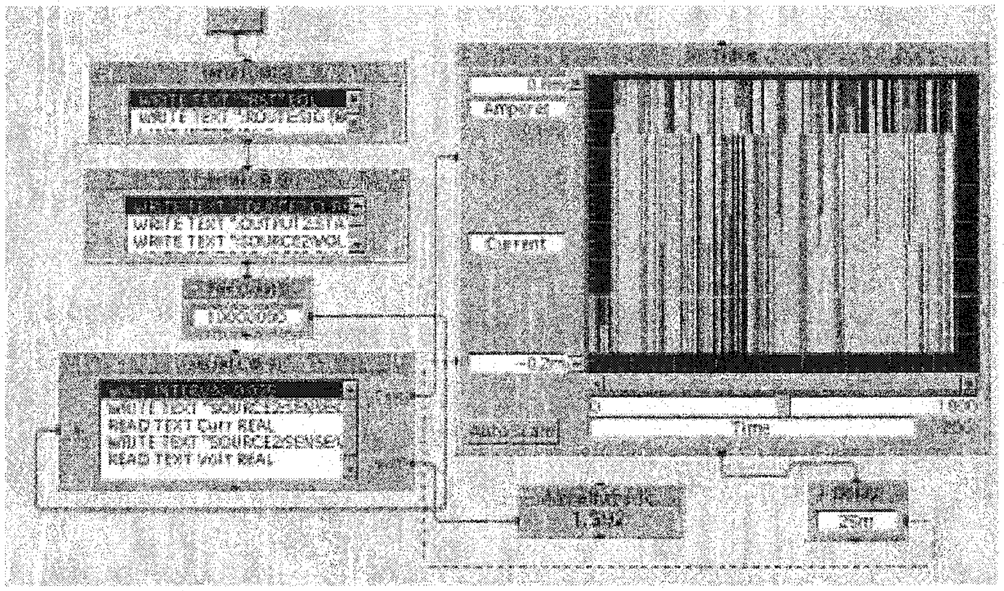


Figure D.7 Current measurement interconnections

Direct I/O objects are used to send commands to the TH1000 Test Head, User Objects and User Functions can be created and saved to generate user-defined test libraries.

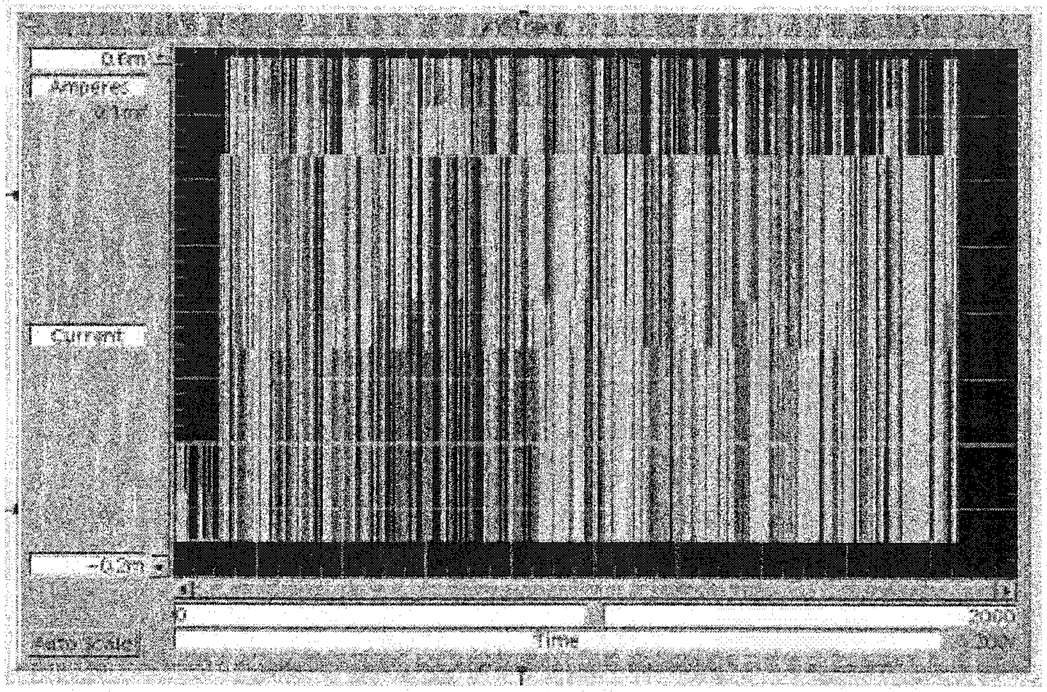


Figure D.8 Current of power supply display

For this test, We set power supply of the core to 1.6 v. Figure D.7 shows VEE test setup for current measurement. Current measurement resolution is 0.2mA. As shown in Figure D.8, the current drawn by the power supply is between 0 and 0.6 mA, which indicate that power consumption of our chip will not exceed 0.6 mA x 1.6 v = 0.96 mW (matches our power estimation of 0.78 mW from the Synopsys Power Compiler)

According to Synopsys Power Compiler, the filters occupy about 80% power consumption of the whole chip. which means our filter power will be less than 0.8 x 0.96 = 0.768mW.

Vita Auctoris

"Hongbo (Jennifer) Li", born October 6, 1968 in Wuhan, HuBei China. Jennifer attended the University of Electronic Science and Technology of China, where she obtained her Honours Bachelor of Engineering in Electrical Engineering in 1990. Jennifer pursued graduate studies at the University of Windsor where she worked on her Master of Applied Science in the area of Electrical and Computer Engineering, focusing on special DSP architectures and VLSI design.